O'REILLY®

3rd Edition

Test-Driven Development with Python

Obey the Testing Goat: Using Django, Selenium, and JavaScript

Harry J.W. Percival

"Testing is essential for developer sanity. Harry does a fantastic job of holding our attention whilst exploring real-world testing practices."
Michael Foord, Python core developer and maintainer of unittest

"This book is far more than an introduction to test-driven development—it's a complete best-practices crash course, from start to finish, on modern web application development with Python."
Kenneth Reitz, fellow at Python Software Foundation

Test-Driven Development with Python

The third edition of this trusted guide shows you how to apply test-driven development (TDD) to building real-world web applications with Python. By writing tests before building each part of your app—and then creating just enough code to pass them—you'll learn how TDD leads to clean, reliable, and maintainable software.

Author Harry J.W. Percival takes you through a practical, end-to-end example of web development using Python 3.14 and Django 5. Along the way, you'll explore tools like Selenium, JavaScript, Git, and mocking, and discover how TDD supports better design decisions, encourages continuous improvement, and instills confidence in your codebase. Whether you're a professional developer or just transitioning into web development, this book offers hands-on experience with modern testing workflows and architecture.

- **Follow the full TDD workflow, from writing tests first to refactoring with confidence**
- **Write unit tests for core logic and functional tests for browser-based interactions**
- **Use mock objects to isolate external systems and simplify integration**
- **Package your application using Docker**
- **Automate deployments and test your code in a staging environment**
- **Validate third-party plug-ins and dependencies within your test suite**
- **Set up continuous integration to run your tests automatically**
- **Enrich your frontend with test-driven JavaScript**

Harry J.W. Percival is a passionate advocate for TDD, sharing his expertise worldwide through talks and workshops. He works for Kraken Technologies, writing software to support the worldwide green energy transition.

PROGRAMMING / PYTHON

US $79.99 CAN $99.99
ISBN: 978-1-098-14871-3

9 781098 148713 57999

O'REILLY®

Praise for *Test-Driven Development with Python*

In this book, Harry takes us on an adventure of discovery with Python and testing. It's an excellent book, fun to read, and full of vital information. It has my highest recommendations for anyone interested in testing with Python, learning Django, or wanting to use Selenium. Testing is essential for developer sanity and it's a notoriously difficult field, full of trade-offs. Harry does a fantastic job of holding our attention whilst exploring real-world testing practices.

—Michael Foord,
Python Core Developer and Maintainer of unittest

This book is far more than an introduction to test-driven development—it's a complete best-practices crash course, from start to finish, into modern web application development with Python. Every web developer needs this book.

—Kenneth Reitz,
Fellow at Python Software Foundation

Harry's book is what we wish existed when we were learning Django. At a pace that's achievable and yet delightfully challenging, it provides excellent instruction for Django and various test practices. The material on Selenium alone makes the book worth purchasing, but there's so much more!

—Daniel and Audrey Roy Greenfeld,
authors of Two Scoops of Django *(Two Scoops Press)*

THIRD EDITION

Test-Driven Development with Python

Obey the Testing Goat: Using Django, Selenium, and JavaScript

Harry J.W. Percival

O'REILLY®

Test-Driven Development with Python
by Harry J.W. Percival

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Brian Guerin
Development Editor: Rita Fernando
Production Editor: Christopher Faucher
Copyeditor: Piper Content Partners
Proofreader: Kim Cofer
Indexer: Ellen Troutman-Zaig
Cover Designer: Susan Brown
Cover Illustrator: Karen Montgomery
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

June 2014: First Edition
August 2017: Second Edition
October 2025: Third Edition

Revision History for the Third Edition
2025-10-30: First Release

See *https://www.oreilly.com/catalog/errata.csp?isbn=0636920873884* for release details.

978-1-098-14871-3

[LSI]

Table of Contents

Part III. Forms and Validation

Part IV. More Advanced Topics in Testing

Preface

This book has been my attempt to share with the world the journey I took from "hacking" to "software engineering". It's mainly about testing, but there's a lot more to it, as you'll soon see.

I want to thank you for reading it.

If you bought a copy, then I'm very grateful. If you're reading the free online version, then I'm *still* grateful that you've decided it's worth spending your time on. Who knows; perhaps once you get to the end, you'll decide it's good enough to buy a physical copy for yourself or a friend.

If you have any comments, questions, or suggestions, I'd love to hear from you. You can reach me directly via *obeythetestinggoat@gmail.com*, or on Mastodon @hjwp. You can also check out the website and my blog.

I hope you'll enjoy reading this book as much as I enjoyed writing it.

Why I Wrote a Book About Test-Driven Development

"Who are you, why have you written this book, and why should I read it?" I hear you ask.

I was lucky enough early on in my career to fall in with a bunch of test-driven development (TDD) fanatics, and it made such a big impact on my programming that I was burning to share it with everyone. You might say I had the enthusiasm of a recent convert, and the learning experience was still a recent memory for me, so that's what led to the first edition, back in 2014.

When I first learned Python (from Mark Pilgrim's excellent *Dive Into Python*), I came across the concept of TDD, and thought, "Yes. I can definitely see the sense in that". Perhaps you had a similar reaction when you first heard about TDD? It seemed like a really sensible approach, a really good habit to get into—like regularly flossing your teeth.

Then came my first big project, and you can guess what happened—there was a client, there were deadlines, there was lots to do, and any good intentions about TDD went straight out of the window.

And, actually, it was fine. I was fine.

At first.

At first I thought I didn't really need TDD because the website was small, and I could easily test whether things worked by just manually checking it out. Click this link *here*, choose that drop-down item *there*, and *this* should happen. Easy. This whole "writing tests" thing sounded like it would have taken *ages*. And besides, I fancied myself, from the full height of my three weeks of adult coding experience, as being a pretty good programmer. I could handle it. Easy.

Then came the fearful goddess Complexity. She soon showed me the limits of my experience.

The project grew. Parts of the system started to depend on other parts. I did my best to follow good principles like DRY (don't repeat yourself), but that just led to some pretty dangerous territory. Soon, I was playing with multiple inheritance. Class hierarchies eight levels deep. `eval` statements.

I became scared of making changes to my code. I was no longer sure what depended on what, and what might happen if I changed this code *over here*…oh gosh, I think that bit over there inherits from it…no, it doesn't; it's overridden. Oh, but it depends on that class variable. Right, well, as long as I override the override it should be fine. I'll just check—but checking was getting much harder. There were lots of sections for the site now, and clicking through them all manually was starting to get impractical. Better to leave well enough alone. Forget refactoring. Just make do.

Soon I had a hideous, ugly mess of code. New development became painful.

Not too long after this, I was lucky enough to get a job with a company called Resolver Systems (now PythonAnywhere), where extreme programming (XP) was the norm. The people there introduced me to rigorous TDD.

Although my previous experience had certainly opened my mind to the possible benefits of automated testing, I still dragged my feet at every stage. "I mean, testing in general might be a good idea, but *really*? All these tests? Some of them seem like a total waste of time…what? Functional tests *as well as* unit tests? Come on, that's overdoing it! And this TDD test/minimal-code-change/test cycle? This is just silly! We don't need all these baby steps! Come on—we can see what the right answer is; why don't we just skip to the end?"

Believe me, I second-guessed every rule, I suggested every shortcut, I demanded justifications for every seemingly pointless aspect of TDD—and I still came out seeing

the wisdom of it all. I've lost count of the number of times I've thought, "Thanks, tests",[1] as a functional test uncovers a regression we would never have predicted, or a unit test saves me from making a really silly logic error. Psychologically, it's made development a much less stressful process. It produces code that's a pleasure to work with.

So, let me tell you *all* about it!

Aims of This Book

My main aim is to impart a methodology—a way of doing web development, which I think makes for better web apps and happier developers. There's not much point in a book that just covers material you could find by googling, so this book isn't a guide to Python syntax, nor a tutorial on web development per se. Instead, I hope to teach you how to use TDD to get more reliably to our shared, holy goal: *clean code that works*.

With that said: I will constantly refer to a real practical example, by building a web app from scratch using tools like Django, Selenium, jQuery, and mocks. I'm not assuming any prior knowledge of any of these, so you should come out the other end of this book with a decent introduction to those tools, as well as the discipline of TDD.

In extreme programming we always pair-program, so I've imagined writing this book as if I was pairing with my previous self, having to explain how the tools work and answer questions about why we code in this particular way. So, if I ever take a bit of a patronising tone, it's because I'm not all that smart, and I have to be very patient with myself. And if I ever sound defensive, it's because I'm the kind of annoying person that systematically disagrees with whatever anyone else says, so sometimes it takes a lot of justifying to convince myself of anything.

1 Thests.

Outline

I've split this book into four parts.

Part I (Chapters 1 to 8): The Basics of TDD and Django
: We dive straight into building a simple web app using TDD. We start by writing a functional test (with Selenium), and then we go through the basics of Django—models, views, templates—with rigorous unit testing at every stage. I also introduce the Testing Goat.

Part II (Chapters 9 to 12): Going to Production
: These chapters are all about deploying your web app to an actual server. We discuss how our tests, and the TDD practice of working incrementally, can take a lot of the pain and risk out of what is normally quite a fraught process.

Part III (Chapters 13 to 16): Forms and Validation
: Here, we get into some of the details of the Django Forms framework, implementing validation, and data integrity using database constraints. We discuss using tests to explore unfamiliar APIs, and the limits of frameworks.

Part IV (Chapters 17 to 27): Advanced Topics in Testing
: Covers some of the more advanced topics in TDD, including spiking (where we relax the rules of TDD temporarily), mocking, working outside-in, and continuous integration (CI).

Now, onto a little housekeeping…

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions

`Constant width`
: Used for program listings and within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords

`Constant width bold`
: Shows commands or other text that should be typed literally by the user

Occasionally I will use the symbol:

```
[...]
```

to signify that some of the content has been skipped, to shorten long bits of output, or to skip down to a relevant section. You will also encounter the following callouts:

This element signifies a tip or suggestion.

This element signifies a general note or aside.

This element indicates a warning or caution.

Submitting Errata

Spotted a mistake or a typo? The sources for this book are available on GitHub, and I'm always very happy to receive issues and pull requests: *https://github.com/hjwp/Book-TDD-Web-Dev-Python*.

Using Code Examples

Code examples are available at *https://github.com/hjwp/book-example*; you'll find branches for each chapter there (e.g., *https://github.com/hjwp/book-example/tree/chapter_03_unit_test_first_view*). You can find a full list and some suggestions on ways of working with this repository in Appendix C.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant

amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Test-Driven Development with Python*, 3rd edition, by Harry J.W. Percival (O'Reilly). Copyright 2026 Harry Percival, 978-1-098-14871-3".

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/TDD-with-python-3e*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Companion Video

I've recorded a 10-part video series to accompany this book.[2] It covers the content of Part I. If you find that you learn well from video-based material, then I encourage you to check it out. Over and above what's in the book, it should give you a feel for what the "flow" of TDD is like, flicking between tests and code and explaining the thought process as we go.

Plus, I'm wearing a delightful yellow t-shirt.

2 The video has not been updated for the third edition, but the content is all mostly the same.

License for the Free Edition

If you're reading the free edition of this book hosted at *http://www.obeythetestinggoat.com*, then the license is Creative Commons Attribution-NonCommercial-NoDerivatives.[3] I want to thank O'Reilly for its fantastic attitude towards licensing; most publishers aren't so forward-thinking.

I see this as a "try before you buy" scheme really. If you're reading this book it's for professional reasons, so I hope that if you like it, you'll buy a copy—if not for yourself, then for a friend! O'Reilly has been great, it deserves your support. You'll find links to buy back on the home page.

3 The no-derivs clause is there because O'Reilly wants to maintain some control over derivative works, but it often does grant permissions for things, so don't hesitate to get in touch if you want to build something based on this book.

Preface to the Third Edition: TDD in the Age of AI

Is there any point in learning TDD now that AI can write code for you? A single prompt could probably generate the entire example application in this book, including all its tests, and the infrastructure config for automated deployment too.

The truth is that it's too early to tell. AI is still in its infancy, and who knows what it'll be able to do in a few years or even months' time.

AI Is Both Insanely Impressive and Incredibly Unreliable

What we do know is that right now, AI is both insanely impressive and incredibly unreliable.

Beyond being able to understand and respond to prompts in normal human language—it's easy to forget how absolutely extraordinary that is; literally science-fiction a few years ago—AI tools can generate working code, they can generate tests, they can help us to break down requirements, brainstorm solutions, quickly prototype new technologies. It's genuinely astonishing.

As we're all finding out though, this all comes with a massive "but". AI outputs are frequently plagued by hallucinations, and in the world of code, that means things that just won't work, even if they look plausible. Worse than that, they can produce code that appears to work, but is full of subtle bugs, security issues, or performance nightmares. From a code quality point of view, we know that AI tools will often produce code that's filled with copy-paste and duplication, weird hacks, and undecipherable spaghetti code that spells a maintenance nightmare.

Mitigations for AI's Shortcomings Sure Look a Lot Like TDD

If you read the advice, even from AI companies themselves, about the best way to work with AI, you'll find that it performs best when working in small, well-defined contexts, with frequent checks for correctness. When taking on larger tasks, the advice is to break them down into smaller, well-defined pieces, with clearly defined success criteria.

When we're thinking about the problem of hallucinations, it sure seems like having a comprehensive test suite and running it frequently, is going to be a must-have.

When we're thinking about code quality, the idea of having a human in the loop, with frequent pauses for review and refactoring, again seems like a key mitigation.

In short, all of the techniques of test-driven development that are outlined in this book:

- Defining a test that describes each small change of functionality, before we write the code for it
- Breaking our problem down into small pieces and working incrementally, with frequent test runs to catch bugs, regressions, and hallucinations
- The "refactor" step in TDD's red/green/refactor cycle, which gives us a regular reminder for the human in the loop to review and improve the code.

TDD is all about finding a structured, safer way of developing software, reducing the risk of bugs and regressions and improving code quality, and these are very much the exact same things that we need to achieve when working with AI.

Leaky Abstractions and the Importance of Experience

"Leaky abstractions" are a diagnosis of a common problem in software development, whereby higher-level abstractions fail in subtle ways, and the complexities of the underlying system leak through.

In the presence of leaky abstractions, you need to understand the lower-level system to be able to work effectively. It's for this reason that, when the switch to third-generation languages (3GLs) happened, programmers who understood the underlying machine code were often the most effective at using the new languages like C and Fortran.

In a similar way, AI offers us a new, higher-level abstraction around writing code, but we can already see the "leaks" in the form of hallucinations and poor code quality. And by analogy to the 3GLs, the programmers who are going to be most effective with AI are going to be the ones who "know what good looks like", both in terms of

code quality, test structure, and so on, but also in terms of what a safe and reliable workflow for software development looks like.

My Own Experiences with AI

In my own experiences of working with AI, I've been very impressed at its ability to write tests, for example… as long as there was already a good first example test to copy from. Its ability to write that *first* test, the one where, as we'll see, a lot of the design (and thinking) happens in TDD, was much more mixed.

Similarly when working in a less "autocomplete" and more "agentic" mode, I saw AI tools do very well on simple problems with clear instructions, but when trying to deal with more complex logic and requirements with ambiguity, I've seen it get dreadfully stuck in loops and dead ends.

When that happened, I found that trying to guide the AI agent back towards taking small steps, working on a single piece at a time, and clarifying requirements in tests, was the best way to get things back on track.

I've also been able to experiment with using the "refactor" step to try and improve the often-terrible code that the AI produced. Here again I had mixed results, where the AI would need a lot of nudging before settling on a solution that felt sufficiently readable and maintainable, to me.

So I'd echo what many others are saying, which is that AI works best when you, the user, are a discerning partner rather than passive recipient.

Ultimately, as software developers, we need to be able to stand by the code we produce, and be accountable for it, no matter what tools were used to write it.

The AI-Enabled Workflow of the Future

The AI-enabled workflow of the future will look very different to what we have now, but all the indications are that the most effective approach is going to be incremental, have checks and balances to avoid hallucinations, and systematically involve humans in the loop to ensure quality.

And the closest workflow we have to that today, is TDD. I'm excited to share it with you!

Prerequisites and Assumptions

Here's an outline of what I'm assuming about you and what you already know, as well as what software you'll need ready and installed on your computer.

Python 3 and Programming

I've tried to write this book with beginners in mind, but if you're new to programming, I'm assuming that you've already learned the basics of Python. So if you haven't already, do run through a Python beginner's tutorial or get an introductory book like *The Quick Python Book* or *Think Python*, or (just for fun) *Invent Your Own Computer Games with Python*—all of which are excellent introductions.

If you're an experienced programmer but new to Python, you should get along just fine. Python is joyously simple to understand.

You should be able to follow this book on Mac, Windows, or Linux. Detailed installation instructions for each OS follow.

This book was tested against Python 3.14. If you're on an earlier version, you will find minor differences in the way things look in my command output listings (tracebacks won't have the ^^^^^^ carets marking error locations, for example), so you're best off upgrading, ideally, if you can.

In any case, I expect you to have access to Python, to know how to launch it from a command line, and to know how to edit a Python file and run it. Again, have a look at the three books I recommended previously if you're in any doubt.

How HTML Works

I'm also assuming you have a basic grasp of how the web works—what HTML is, what a POST request is, and so on. If you're not sure about those, you'll need to find a basic HTML tutorial; there are a few at *https://developer.mozilla.org/Learn_web_development*. If you can figure out how to create an HTML page on your PC and look at it in your browser, and understand what a form is and how it might work, then you're probably OK.

Django

The book uses the Django framework, which is (probably) the most well-established web framework in the Python world. I've written this book assuming that the reader has no prior knowledge of Django, but if you're new to Python *and* new to web development *and* new to testing, you may occasionally find that there's just one too many topics and sets of concepts to try and take on board. If that's the case, I recommend taking a break from the book, and taking a look at a Django tutorial. DjangoGirls is the best, most beginner-friendly tutorial I know of. Django's official tutorial is also excellent for more experienced programmers.

JavaScript

There's a little bit of JavaScript in the second half of the book. If you don't know JavaScript, don't worry about it until then. And if you find yourself a little confused, I'll recommend a couple of guides at that point.

Read on for installation instructions.

Required Software Installations

Aside from Python, you'll need:

The Firefox web browser
: Selenium can actually drive any of the major browsers, but I chose Firefox because it's the least in hock to corporate interests.

The Git version control system
: This is available for any platform, at *http://git-scm.com*. On Windows, it comes with the *bash* command line, which is needed for the book. See "Windows Notes" on page xxxii.

A virtualenv with Python 3.14, Django 5.2, and Selenium 4 in it
: Python's virtualenv and pip tools now come bundled with Python (they didn't always used to, so this is a big hooray). Detailed instructions for preparing your virtualenv follow.

MacOS Notes

MacOS installations for Python and Git are relatively straightforward:

- Python 3.14 should install without a fuss from its downloadable installer. It will automatically install `pip`, too.
- Git's installer should also "just work".
- You might also want to check out Homebrew. It's a fairly reliable way of installing common Unix tools on a Mac.[1] Although the normal Python installer is now fine, you may find Homebrew useful in future. It does require you to download all 1.1 GB of Xcode, but that also gives you a C compiler, which is a useful side effect.
- If you want to run multiple different versions of Python on your Mac, tools like uv or pyenv can help. The downside is that each is one more fiddly tool to have to learn. But the key is to make sure, when creating your virtualenv, that you use the right version of Python. From then on, you shouldn't need to worry about it, at least not when following this book.

Similarly to Windows, the test for all this is that you should be able to open a terminal and just run `git`, `python3`, or `pip` from anywhere. If you run into any trouble, the search terms "system path" and "command not found" should provide good troubleshooting resources.

1 I wouldn't recommend installing Firefox via Homebrew though: `brew` puts the Firefox binary in a strange location, and it confuses Selenium. You can work around it, but it's simpler to just install Firefox in the normal way.

Linux Notes

If you're on Linux, I'm assuming you're already a glutton for punishment, so you don't need detailed installation instructions. But in brief, if Python 3.14 isn't available directly from your package manager, you can try the following:

- On Ubuntu, you can install the deadsnakes PPA. Make sure you `apt install python3.14-venv` as well as just `python3.14` to un-break the default Debian version of Python.
- Alternatively, uv and pyenv both let you manage multiple Python versions on the same machine, but it is yet another thing to have to learn and remember.
- Alternatively, compiling Python from source is actually surprisingly easy!

However you install it, make sure you can run Python 3.14 from a terminal.

Windows Notes

Windows users can sometimes feel a little neglected in the open source world. As macOS and Linux are so prevalent, it's easy to forget there's a world outside the Unix paradigm. Backslashes as directory separators? Drive letters? What? Still, it is absolutely possible to follow along with this book on Windows. Here are a few tips:

- When you install Git for Windows, it will include "Git Bash". Use this as your main command prompt throughout the book, and you'll get all the useful GNU command-line tools like `ls`, `touch`, and `grep`, plus forward-slash directory separators.
- During the Git installation, you'll get the option to choose the default editor used by Git. Unless you're already a Vim user (or are desperate to learn), I'd suggest using a more familiar editor—even just Notepad! See Figure P-1.
- Also in the Git installer, choose "Use Windows' default console"; otherwise, Python won't work properly in the Git Bash window.
- When you install Python, tick the option that says "Add python.exe to PATH" as in Figure P-2, so that you can easily run Python from the command line.

The test for all this is that you should be able to go to a Git Bash command prompt and just run `python` or `pip` from any folder.

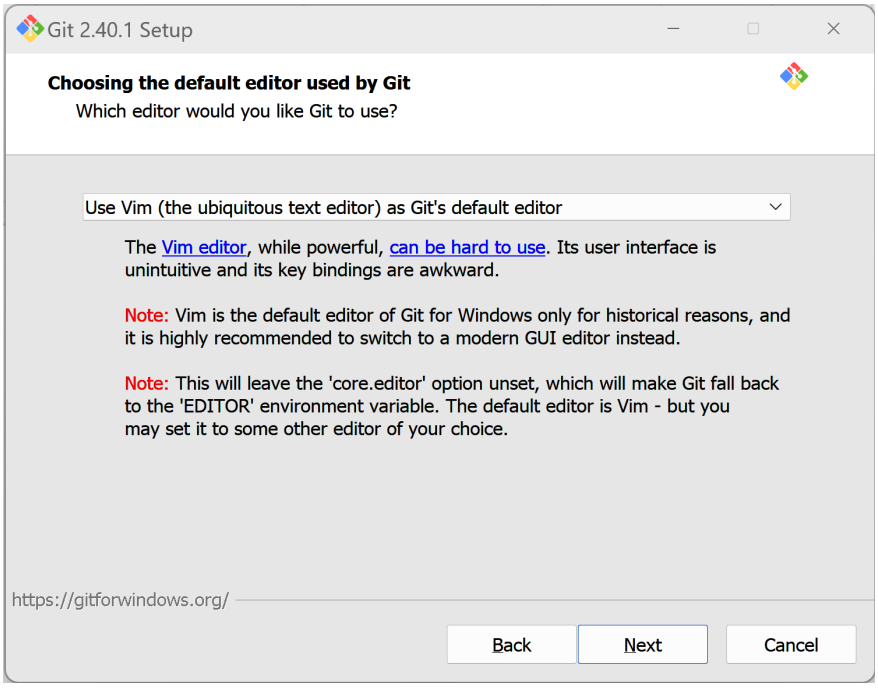


Figure P-1. Choose a nice default editor for Git

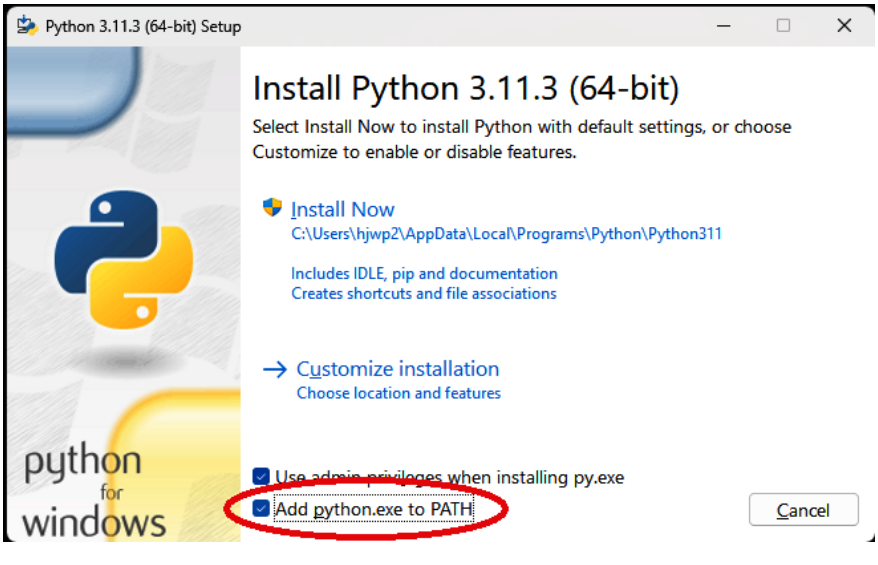


Figure P-2. Add Python to the system path from the installer

Installing Firefox

Firefox is available as a download for Windows and macOS from *firefox.com*. On Linux, you probably already have it installed, but otherwise your package manager will have it.

Make sure you have the latest version, so that the "geckodriver" browser automation module is available.

Setting Up Your Virtualenv

A Python virtualenv (short for virtual environment) is how you set up your environment for different Python projects. It enables you to use different packages (e.g., different versions of Django, and even different versions of Python) in each project. And because you're not installing things system-wide, it means you don't need root permissions.

Let's create a virtualenv. I'm assuming you're working in a folder called *goat-book*, but you can name your work folder whatever you like. Stick to the name ".venv" for the virtualenv, though:

```
$ cd goat-book
$ py -3.14 -m venv .venv
```

On Windows, the `py` executable is a shortcut for different Python versions. On Mac or Linux, we use `python3.14`:

```
$ cd goat-book
$ python3.14 -m venv .venv
```

Activating and Deactivating the Virtualenv

Whenever you're working through the book, you'll want to make sure your virtualenv has been "activated". You can always tell when your virtualenv is active because, in your prompt, you'll see `(.venv)` in parentheses. But you can also check by running `which python` to check whether Python is currently the system-installed one or the virtualenv one.

The command to activate the virtualenv is `source .venv/Scripts/activate` on Windows and `source .venv/bin/activate` on Mac/Linux. The command to deactivate is just `deactivate`.

Try it out like this, on Windows:

```
$ source .venv/Scripts/activate
(.venv)$
(.venv)$ which python
/C/Users/harry/goat-book/.venv/Scripts/python
(.venv)$ deactivate
$
$ which python
/c/Users/harry/AppData/Local/Programs/Python/Python312-32/python
```

Or like this, on Mac/Linux:

```
$ source .venv/bin/activate
(.venv)$
(.venv)$ which python
/home/harry/goat-book/.venv/bin/python
(.venv)$ deactivate
$
$ which python
/usr/bin/python
```

Always make sure your virtualenv is active when working on the book. Look out for the `(.venv)` in your prompt, or run `which python` to check.

Virtualenvs and IDEs

If you're using an IDE like PyCharm or Visual Studio Code, you should be able to configure them to use the virtualenv as the default Python interpreter for the project.

You should then be able to launch a terminal inside the IDE with the virtualenv already activated.

Installing Django and Selenium

We'll install Django 5.2 and the latest Selenium.[2] Remember to make sure your virtualenv is active first!

2 You might be wondering why I'm not mentioning a specific version of Selenium. It's because Selenium is constantly being updated to keep up with changes in web browsers, and as we can't really pin our browser to a specific version, we're best off using the latest Selenium. It was version 4.24 at the time of writing.

```
(.venv) $ pip install "django<6" "selenium"
Collecting django<6
  Downloading Django-5.2.3-py3-none-any.whl (8.0 MB)
     ---------------------------------------- 8.1/8.1 MB 7.6 MB/s eta 0:00:00
Collecting selenium
  Downloading selenium-4.24.0-py3-none-any.whl (6.5 MB)
     ---------------------------------------- 6.5/6.5 MB 6.3 MB/s eta 0:00:00
Collecting asgiref>=3.8.1 (from django<6)
  Downloading asgiref-3.8.1-py3-none-any.whl.metadata (9.3 kB)
Collecting sqlparse>=0.3.1 (from django<6)Collecting sqlparse>=0.3.1 (from
django<6)
  [...]
Installing collected packages: sortedcontainers, websocket-client, urllib3,
typing_extensions, sqlparse, sniffio, pysocks, idna, h11, certifi, attrs,
asgiref, wsproto, outcome, django, trio, trio-websocket, selenium
Successfully installed asgiref-3.8.1 attrs-25.3.0 certifi-2025.4.26
django-5.2.3 [...]
selenium-4.32.0 [...]
```

Check that it works:

```
(.venv) $ python -c "from selenium import webdriver; webdriver.Firefox()"
```

This should pop open a Firefox web browser, which you'll then need to close.

If you see an error, you'll need to debug it before you go further. On Linux/Ubuntu, I ran into a bug, which needs to be fixed by setting an environment variable called `TMPDIR`.

Some Error Messages You're Likely to See When You Inevitably Fail to Activate Your Virtualenv

If you're new to virtualenvs—or even if you're not, to be honest—at some point you're *guaranteed* to forget to activate it, and then you'll be staring at an error message. Happens to me all the time. Here are some of the things to look out for:

```
ModuleNotFoundError: No module named 'selenium'
```

Or:

```
ModuleNotFoundError: No module named 'django'
[...]
ImportError: Couldn't import Django. Are you sure it's installed and available
on your PYTHONPATH environment variable? Did you forget to activate a virtual
environment?
```

As always, look out for that `(.venv)` in your command prompt, and a quick `source .venv/Scripts/activate` or `source .venv/bin/activate` is probably what you need to get it working again.

Here's another, for good measure:

```
bash: .venv/Scripts/activate: No such file or directory
```

This means you're not currently in the right directory for working on the project. Try a `cd goat-book`, or similar.

Alternatively, if you're sure you're in the right place, you may have run into a bug from an older version of Python, where it wouldn't install an activate script that was compatible with Git Bash. Reinstall Python 3, and make sure you have version 3.6.3 or later, and then delete and re-create your virtualenv.

If you see something like this, it's probably the same issue and you need to upgrade Python:

```
bash: @echo: command not found
bash: .venv/Scripts/activate.bat: line 4:
      syntax error near unexpected token `(
bash: .venv/Scripts/activate.bat: line 4: `if not defined PROMPT ('
```

Final one! Consider this:

```
'source' is not recognized as an internal or external command,
operable program or batch file.
```

If you see this, it's because you've launched the default Windows command prompt, `cmd`, instead of Git Bash. Close it and open the latter.

On Anaconda

Anaconda is another tool for managing different Python environments. It's particularly popular on Windows and for scientific computing, where it can be hard to get some of the compiled libraries to install.

In the world of web programming, it's much less necessary, so *I recommend you do not use Anaconda for this book*.

Happy coding!

Did these instructions not work for you? Or have you got better ones? Get in touch: *obeythetestinggoat@gmail.com*!

Acknowledgments

Lots of people to thank, without whom this book would never have happened, and/or would have been even worse than it is.

Thanks first to "Greg" at `$OTHER_PUBLISHER`, who was the first person to encourage me to believe it really could be done. Even though your employers turned out to have overly regressive views on copyright, I'm forever grateful that you believed in me.

Thanks to Michael Foord, another ex-employee of Resolver Systems, for providing the original inspiration by writing a book himself, and thanks for his ongoing support for the project. Thanks also to my boss Giles Thomas, for foolishly allowing another one of his employees to write a book (although I believe he's now changed the standard employment contract to say "no books"). Thanks also for your ongoing wisdom and for setting me off on the testing path.

Thanks to my other colleagues, Glenn Jones and Hansel Dunlop, for being invaluable sounding boards, and for your patience with my one-track-record conversation over the last year.

Thanks to my wife, Clementine, and to both my families—without whose support and patience I would never have made it. I apologise for all the time spent with my nose in the computer on what should have been memorable family occasions. I had no idea when I set out what the book would do to my life ("Write it in my spare time, you say? That sounds reasonable…"). I couldn't have done it without you.

Thanks to my tech reviewers, Jonathan Hartley, Nicholas Tollervey, and Emily Bache, for your encouragements and invaluable feedback. Especially Emily, who actually conscientiously read every single chapter. Partial credit to Nick and Jon, but that should still be read as eternal gratitude. Having y'all around made the whole thing less of a lonely endeavour. Without all of you, the book would have been little more than the nonsensical ramblings of an idiot.

Thanks to everyone else who's given up their time to give some feedback on the book, out of nothing more than the goodness of their heart: Gary Bernhardt, Mark Lavin,

Matt O'Donnell, Michael Foord, Hynek Schlawack, Russell Keith-Magee, Andrew Godwin, Kenneth Reitz, and Nathan Stocks. Thanks for being much smarter than I am, and for preventing me from saying several stupid things. Naturally, there are still plenty of stupid things left in the book, for which y'all can absolutely not be held responsible.

Thanks to my editor, Meghan Blanchette, for being a very friendly and likeable slave driver, and for keeping the book on track, both in terms of timescales and by restraining my sillier ideas. Thanks to all the others at O'Reilly for your help, including Sarah Schneider, Kara Ebrahim, and Dan Fauxsmith for letting me keep British English. Thanks to Charles Roumeliotis for your help with style and grammar. We may never see eye-to-eye on the merits of Chicago School quotation/punctuation rules, but I sure am glad you were around. And thanks to the design department for giving us a goat for the cover!

And thanks most especially to all my early release readers, for all your help picking out typos, for your feedback and suggestions, for all the ways in which you helped to smooth out the learning curve in the book, and most of all for your kind words of encouragement and support that kept me going. Thank you Jason Wirth, Dave Pawson, Jeff Orr, Kevin De Baere, crainbf, dsisson, Galeran, Michael Allan, James O'Donnell, Marek Turnovec, SoonerBourne, julz, Cody Farmer, William Vincent, Trey Hunner, David Souther, Tom Perkin, Sorcha Bowler, Jon Poler, Charles Quast, Siddhartha Naithani, Steve Young, Roger Camargo, Wesley Hansen, Johansen Christian Vermeer, Ian Laurain, Sean Robertson, Hari Jayaram, Bayard Randel, Konrad Korżel, Matthew Waller, Julian Harley, Barry McClendon, Simon Jakobi, Angelo Cordon, Jyrki Kajala, Manish Jain, Mahadevan Sreenivasan, Konrad Korżel, Deric Crago, Cosmo Smith, Markus Kemmerling, Andrea Costantini, Daniel Patrick, Ryan Allen, Jason Selby, Greg Vaughan, Jonathan Sundqvist, Richard Bailey, Diane Soini, Dale Stewart, Mark Keaton, Johan Wärlander, Simon Scarfe, Eric Grannan, Marc-Anthony Taylor, Maria McKinley, John McKenna, Rafał Szymański, Roel van der Goot, Ignacio Reguero, TJ Tolton, Jonathan Means, Theodor Nolte, Jungsoo Moon, Craig Cook, Gabriel Ewilazarus, Vincenzo Pandolfo, David "farbish2", Nico Coetzee, Daniel Gonzalez, Jared Contrascere, Zhao 赵亮, and many, many more. If I've missed your name, you have an absolute right to be aggrieved; I am incredibly grateful to you too, so write to me and I will try and make it up to you in any way I can.

And finally thanks to you, the latest reader, for deciding to check out the book! I hope you enjoy it.

Additional Thanks for the Second Edition

Thanks to my wonderful editor for the second edition, Nan Barber, and to Susan Conant, Kristen Brown, and the whole team at O'Reilly. Thanks once again to Emily and Jonathan for tech reviewing, as well as to Edward Wong for his very thorough notes. Any remaining errors and inadequacies are all my own.

Thanks also to the readers of the free edition who contributed comments, suggestions, and even some pull requests. I have definitely missed some of you on this list, so apologies if your name isn't here, but thanks to Emre Gonulates, Jésus Gómez, Jordon Birk, James Evans, Iain Houston, Jason DeWitt, Ronnie Raney, Spencer Ogden, Suresh Nimbalkar, Darius, Caco, LeBodro, Jeff, Duncan Betts, wasabigeek, joegnis, Lars, Mustafa, Jared, Craig, Sorcha, TJ, Ignacio, Roel, Justyna, Nathan, Andrea, Alexandr, bilyanhadzhi, mosegontar, sfarzy, henziger, hunterji, das-g, juanriaza, GeoWill, Windsooon, gonulate, Margie Roswell, Ben Elliott, Ramsay Mayka, peterj, 1hx, Wi, Duncan Betts, Matthew Senko, Neric "Kasu" Kaz, Dominic Scotto, Andrey Makarov, and many, many more.

Additional Thanks for the Third Edition

Thanks to my editor, Rita Fernando, thanks to my tech reviewers Béres Csanád, David Seddon, Sebastian Buczyński, and Jan Giacomelli, and thanks to all the early release readers for your feedback, big and small, including Jonathan H., James Evans, Patrick Cantwell, Devin Schumacher, Nick Nielsen, Teemu Viikeri, Andrew Zipperer, artGonza, Joy Denebeim, mshob23, Romilly Cocking, Zachary Kerbo, Stephanie Goulet, David Carter, Jim Win Man, Alex Kennett, Ivan Schneider, Lars Berberich, Rodrigo Jacznik, Tom Nguyen, rokbot, Nikita Durne, and to anyone I've missed off this list, my sincere apologies, ping me and I'll add you, and thank you thank you once again.

Extra Thanks for Csanàd

Every single one of the tech reviewers for this edition was invaluable, and they all contributed in different and complementary ways.

But I want to give extra thanks to Csanàd, who went beyond the normal remit of a tech reviewer, so far as to do substantial actual rewrites of several chapters in Part III.

You can't blame him for anything in there though, because I've been over them since, so any errors or problems you might spot are definitely things I've added since.

Anyways, thanks so much Csanàd, you helped me feel like I wasn't entirely alone.

PART I

The Basics of TDD and Django

In this first part, I'm going to introduce the basics of *test-driven development* (TDD). We'll build a real web application from scratch, writing tests first at every stage.

We'll cover functional testing with Selenium, as well as unit testing, and see the difference between the two. I'll introduce the TDD workflow, red/green/refactor.

I'll also be using a version control system (Git). We'll discuss how and when to do commits and integrate them with the TDD and web development workflow.

We'll be using Django, the Python world's most popular web framework (probably). I've tried to introduce the Django concepts slowly and one at a time, and provide lots of links to further reading. If you're a total beginner to Django, I thoroughly recommend taking the time to read them. If you find yourself feeling a bit lost, take a couple of hours to go through the official Django tutorial and then come back to the book.

In Part I, you'll also get to meet the Testing Goat…

Be careful with copy and paste. If you're working from a digital version of the book, it's natural to want to copy and paste code listings from the book as you're working through it. It's much better if you don't: typing things in by hand gets them into your muscle memory, and just feels much more real. You also inevitably make the occasional typo, and debugging is an important thing to learn.

Quite apart from that, you'll find that the quirks of the PDF format mean that weird stuff often happens when you try to copy/paste from it…

CHAPTER 1

Getting Django Set Up Using a Functional Test

Test-driven development isn't something that comes naturally. It's a discipline, like a martial art, and just like in a Kung Fu movie, you need a bad-tempered and unreasonable master to force you to learn the discipline. Ours is the Testing Goat.

Obey the Testing Goat! Do Nothing Until You Have a Test

The Testing Goat is the unofficial mascot[1] of TDD in the Python testing community. It probably means different things to different people, but, to me, the Testing Goat is a voice inside my head that keeps me on the True Path of Testing—like one of those little angels or demons that pops up by your shoulder in the cartoons, but with a very niche set of concerns. I hope, with this book, to install the Testing Goat inside your head too.

So we've decided to build a web app, even if we're not quite sure what it's going to do yet. Normally, the first step in web development is getting your web framework installed and configured. *Download this, install that, configure the other, run the script*…but TDD requires a different mindset. When you're doing TDD, you always have the Testing Goat inside your head—single-minded as goats are—bleating "Test first, test first!"

In TDD the first step is always the same: *write a test*.

1 OK more of a minor running joke from PyCon in the mid 2010s, which I am single-handedly trying to make into a thing.

First we write the test; *then* we run it and check that it fails as expected. *Only then* do we go ahead and build some of our app. Repeat that to yourself in a goat-like voice. I know I do.

Another thing about goats is that they take one step at a time. That's why they seldom fall off things, see, no matter how steep they are—as you can see in Figure 1-1.

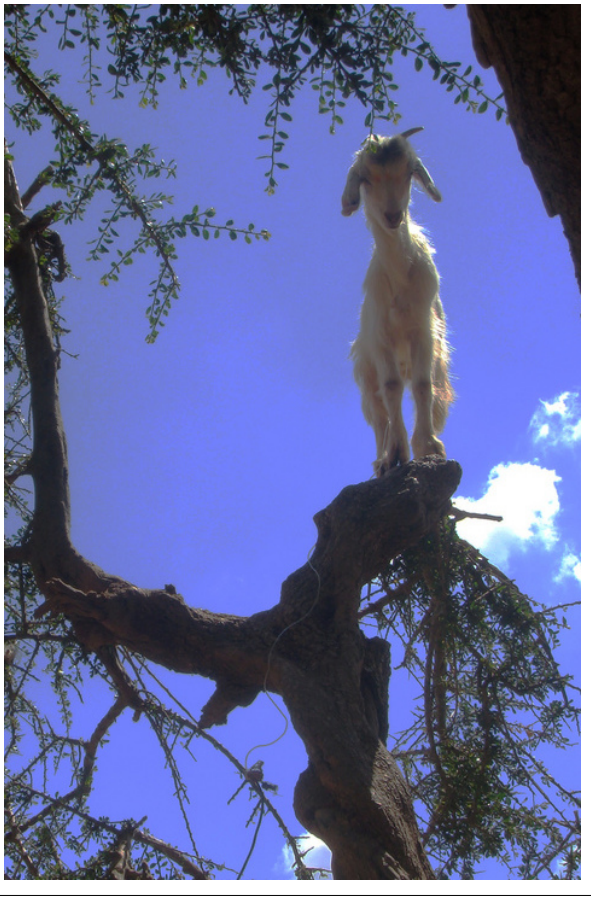


Figure 1-1. Goats are more agile than you think (source: Caitlin Stewart, on Flickr)

We'll proceed with nice small steps; we're going to use *Django*, which is a popular Python web framework, to build our app.

The first thing we want to do is check that we've got Django installed and that it's ready for us to work with. The *way* we'll check is by confirming that we can spin up Django's development server and actually see it serving up a web page, in our web browser, on our local computer. We'll use the *Selenium* browser automation tool for this.

Create a new Python file called *functional_tests.py* wherever you want to keep the code for your project, and enter the following code. If you feel like making a few little goat noises as you do it, it may help:

functional_tests.py

```
from selenium import webdriver

browser = webdriver.Firefox()
browser.get("http://localhost:8000")

assert "Congratulations!" in browser.title
print("OK")
```

That's our first *functional test* (FT); I'll talk more about what I mean by functional tests, and how they contrast with unit tests, in a bit. For now, it's enough to assure ourselves that we understand what it's doing:

- Starting a Selenium WebDriver to pop up a real Firefox browser window.
- Using it to open up a web page, which we're expecting to be served from the local computer.
- Checking (making a test assertion) that the page has the word "Congratulations!" in its title.
- If all goes well, we print OK.

Let's try running it:

```
$ python functional_tests.py
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 4, in <module>
    browser.get("http://localhost:8000")
  File ".../selenium/webdriver/remote/webdriver.py", line 483, in get
    self.execute(Command.GET, {"url": url})
  File ".../selenium/webdriver/remote/webdriver.py", line 458, in execute
    self.error_handler.check_response(response)
  File ".../selenium/webdriver/remote/errorhandler.py", line 232, in
check_response
    raise exception_class(message, screen, stacktrace)
selenium.common.exceptions.WebDriverException: Message: Reached error page: abo
ut:neterror?e=connectionFailure&u=http%3A//localhost%3A8000/[...]
Stacktrace:
RemoteError@chrome://remote/content/shared/RemoteError.sys.mjs:8:8
WebDriverError@chrome://remote/content/shared/webdriver/Errors.sys.mjs:182:5
UnknownError@chrome://remote/content/shared/webdriver/Errors.sys.mjs:530:5
[...]
```

You should see a browser window pop up trying to open *localhost:8000*, showing the "Unable to connect" error page. If you switch back to your console, you'll see the big, ugly error message telling us that Selenium ran into an error page. And then, you will probably be irritated at the fact that it left the Firefox window lying around your desktop for you to tidy up. We'll fix that later!

If, instead, you see an error trying to import Selenium, or an error trying to find something called "geckodriver", you might need to go back and have another look at the "Prerequisites and Assumptions" section.

What to Do If You Get a Firefox Upgrade Pop-up

Now and again, when running Selenium tests, you might encounter a strange pop-up window, such as the one shown in Figure 1-2.

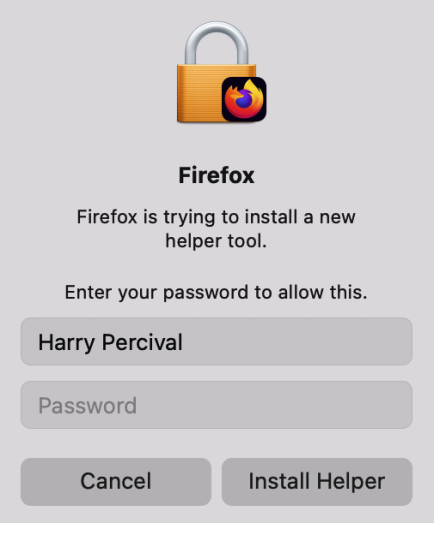


Figure 1-2. Firefox wants to install a new what now?

This happens when Firefox has automatically downloaded a new version, in the background. When Selenium tries to load a fresh Firefox session, it wants to install the latest version of its "geckodriver" plugin.

To resolve the issue, you have to close the Selenium browser window, go back to your main browser window and tell it to install the upgrade and restart itself, and then try again.

If something strange is going on with your FTs, it's worth checking if there's a Firefox upgrade pending.

For now though, we have a *failing test*, so that means we're allowed to start building our app.

Getting Django Up and Running

As you've definitely read "Prerequisites and Assumptions" by now, you've already got Django installed (right?). The first step in getting Django up and running is to create a *project*, which will be the main container for our site. Django provides a little command-line tool for this:

```
$ django-admin startproject superlists .
```

Don't forget that "." at the end; it's important!

That will create a file called *manage.py* in your current folder, and a subfolder called "*superlists*", with more stuff inside it:

```
.
├── functional_tests.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Make sure your project folder looks exactly like this! If you see two nested folders called "superlists", it's because you forgot the "." in the command. Delete them and try again, or there will be lots of confusion with paths and working directories.

The *superlists* folder is intended for stuff that applies to the whole project—like *settings.py*, which is used to store global configuration information for the site.

But the main thing to notice is *manage.py*. That's Django's Swiss Army knife, and one of the things it can do is run a development server. Let's try that now:

```
$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
March 17, 2023 - 18:07:30
Django version 5.2.4, using settings 'superlists.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

That's Django's development server now up and running on our machine.

It's safe to ignore that message about "unapplied migrations" for now. We'll look at migrations in Chapter 5.

Leave it there and open another command shell. Navigate to your project folder, activate your virtualenv, and then try running our test again:

```
$ python functional_tests.py
OK
```

Not much action on the command line, but you should notice two things: firstly, there was no ugly `AssertionError` and, secondly, the Firefox window that Selenium popped up had a different-looking page on it.

If you see an error saying "ModuleNotFoundError: No module named selenium", you've forgotten to activate your virtualenv. Check the "Prerequisites and Assumptions" section again, if you need to.

Well, it may not look like much, but that was our first ever passing test! Hooray!

If it all feels a bit too much like magic, like it wasn't quite real, why not go and take a look at the dev server manually, by opening a web browser yourself and visiting *http://localhost:8000*? You should see something like Figure 1-3.

You can quit the development server now if you like, back in the original shell, using Ctrl+C.

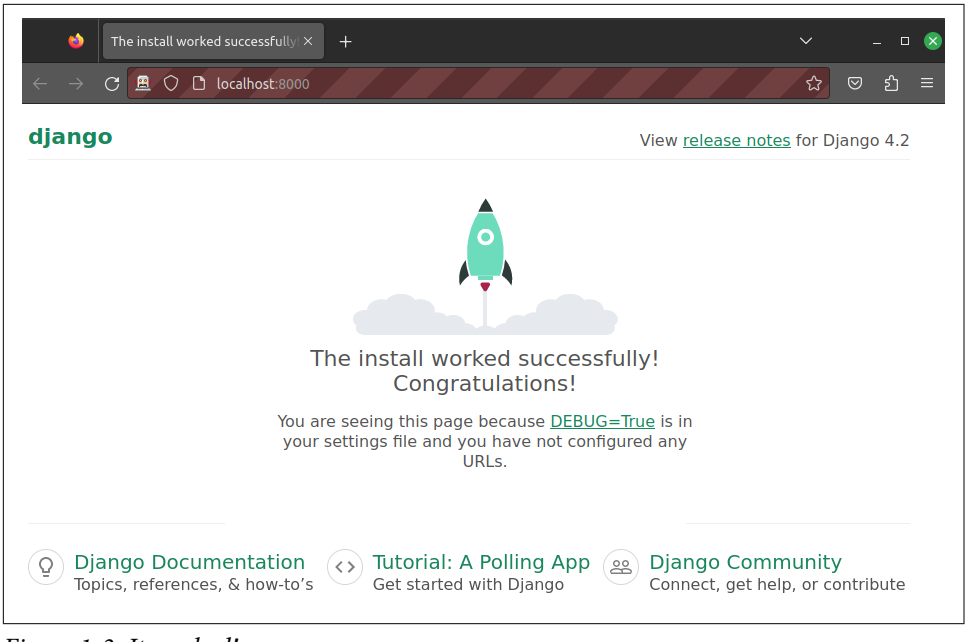


Figure 1-3. It worked!

Adieu to Roman Numerals!

So many introductions to TDD use Roman numerals in their examples that it has become a running joke—I even started writing one myself. If you're curious, you can find it on my GitHub page.

Roman numerals, as an example, are both good and bad. It's a nice "toy" problem, reasonably limited in scope, and you can explain the core of TDD quite well with it.

The problem is that it can be hard to relate to the real world. That's why I've decided to use the building of a real web app, starting from nothing, as my example. Although it's a simple web app, my hope is that it will be easier for you to carry across to your next real project.

In addition, it means we can start out using functional tests as well as unit tests, and demonstrate a TDD workflow that's more like real life, and less like that of a toy project.

Starting a Git Repository

There's one last thing to do before we finish the chapter: start to commit our work to a *version control system* (VCS). If you're an experienced programmer, you don't need to hear me preaching about version control. But if you're new to it, please believe me when I say that VCS is a must-have. As soon as your project gets to be more than a few weeks old and a few lines of code, having a tool available to look back over old versions of code, revert changes, explore new ideas safely, even just as a backup…It's hard to overstate how useful that is. TDD goes hand in hand with version control, so I want to make sure I impart how it fits into the workflow.

Our Working Directory Is Always the Folder That Contains manage.py

We'll be using this same folder throughout the book as our working directory—if in doubt, it's the one that contains *manage.py*.

(For simplicity, in my command listings, I'll always show it as: *…goat-book/*. Although it will probably actually be something like: */home/kind-reader-username/my-python-projects/goat-book/*.)

Whenever I show a command to type in, I will assume we're in this directory. Similarly, if I mention a path to a file, it will be relative to this directory. So, for example, *superlists/settings.py* means the *settings.py* inside the *superlists* folder.

So, our first commit! If anything, it's a bit late; shame on us. We're using *Git* as our VCS, 'cos it's the best.

Let's start by doing the `git init` to start the repository:

```
$ ls
db.sqlite3  functional_tests.py  manage.py  superlists

$ git init .
Initialized empty Git repository in ...goat-book/.git/
```

Setting the Default Branch Name in Git

If you see this message:

```
hint: Using 'master' as the name for the initial branch. This default branch
hint: name is subject to change. To configure the initial branch name to use
hint: in all of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in ...goat-book/.git/
```

Consider following the advice and choosing an explicit default branch name. I chose `main`. It's a popular choice, and you might see it here and there in the book. So if you want to match that, do:

```
$ git config --global init.defaultBranch main
# then let's re-create our git repo by deleting and starting again:
$ rm -rf .git
$ git init .
Initialized empty Git repository in ...goat-book/.git/
```

Now let's take a look and see what files we want to commit:

```
$ ls
db.sqlite3 functional_tests.py manage.py superlists
```

There are a few things in here that we *don't* want under version control: *db.sqlite3* is the database file, and our virtualenv shouldn't be in Git either. We'll add all of them to a special file called *.gitignore* which, um, tells Git what to ignore:

```
$ echo "db.sqlite3" >> .gitignore
$ echo ".venv" >> .gitignore
```

Next we can add the rest of the contents of the current "." folder:

```
$ git add .
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   functional_tests.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/__pycache__/__init__.cpython-314.pyc
        new file:   superlists/__pycache__/settings.cpython-314.pyc
        new file:   superlists/__pycache__/urls.cpython-314.pyc
        new file:   superlists/__pycache__/wsgi.cpython-314.pyc
        new file:   superlists/asgi.py
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py
```

Oops! We've got a bunch of *.pyc* files in there; it's pointless to commit those. Let's remove them from Git and add them to *.gitignore* too:

```
$ git rm -r --cached superlists/__pycache__
rm 'superlists/__pycache__/__init__.cpython-314.pyc'
rm 'superlists/__pycache__/settings.cpython-314.pyc'
rm 'superlists/__pycache__/urls.cpython-314.pyc'
rm 'superlists/__pycache__/wsgi.cpython-314.pyc'
$ echo "__pycache__" >> .gitignore
$ echo "*.pyc" >> .gitignore
```

Now let's see where we are…

```
$ git status
On branch main

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   .gitignore
        new file:   functional_tests.py
        new file:   manage.py
        new file:   superlists/__init__.py
        new file:   superlists/asgi.py
        new file:   superlists/settings.py
        new file:   superlists/urls.py
        new file:   superlists/wsgi.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

        modified:   .gitignore
```

You'll see I'm using `git status` a lot—so much so that I often alias it to `git st`…I'm not telling you how to do that though; I leave you to discover the secrets of Git aliases on your own!

Looking good—we're ready to do our first commit!

```
$ git add .gitignore
$ git commit
```

When you type `git commit`, it will pop up an editor window for you to write your commit message in. Mine looked like Figure 1-4.[2]

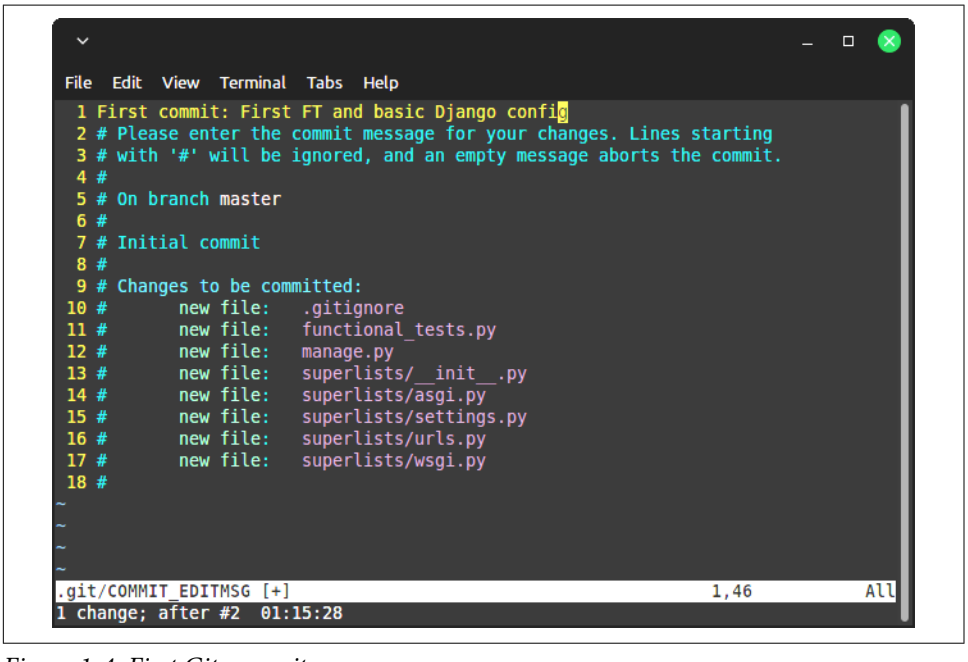


Figure 1-4. First Git commit

If you want to really go to town on Git, this is the time to also learn about how to push your work to a cloud-based VCS hosting service like GitHub or GitLab. They'll be useful if you think you want to follow along with this book on different computers. I leave it to you to find out how they work; they have excellent documentation. Alternatively, you can wait until Chapter 25, where we'll use one.

That's it for the VCS lecture. Congratulations! You've written a functional test using Selenium, and you've gotten Django installed and running, in a certifiable, test-first, goat-approved TDD way. Give yourself a well-deserved pat on the back before moving on to Chapter 2.

2 Did a strange terminal-based editor (the dreaded Vim) pop up and you had no idea what to do? Or did you see a message about account identity and `git config --global user.username`? Check out the Git manual and its basic configuration section. PS: To quit Vim, it's Esc, then `:q!`

CHAPTER 2

Extending Our Functional Test Using the unittest Module

Let's adapt our test, which currently checks for the default Django "it worked" page, and check instead for some of the things we want to see on the real front page of our site.

Time to reveal what kind of web app we're building: a to-do lists site! I know, I know, every other web dev tutorial online is also a to-do lists app, or maybe a blog or a polls app. I'm very much following fashion.

The reason is that a to-do list is a really nice example. At its most basic, it is very simple indeed—just a list of text strings—so it's easy to get a "minimum viable" list app up and running. But it can be extended in all sorts of ways—different persistence models, adding deadlines, reminders, sharing with other users, and improving the client-side UI. There's no reason to be limited to just "to-do" lists either; they could be any kind of lists. But the point is that it should enable me to demonstrate all of the main aspects of web programming, and how you apply TDD to them.

Using a Functional Test to Scope Out a Minimum Viable App

Tests that use Selenium let us drive a real web browser, so they really let us see how the application *functions* from the user's point of view. That's why they're called *functional tests*.

This means that an FT can be a sort of specification for your application. It tends to track what you might call a *user story*, and follows how the user might work with a particular feature and how the app should respond to them.[1]

Terminology: Functional Test == End-to-End Test == Acceptance Test

What I call functional tests, some people prefer to call *end-to-end tests*, or, slightly less commonly, *system tests*.

The main point is that these kinds of tests look at how the whole application functions, from the outside. Another name is *black box test*, or *closed box test*, because the test doesn't know anything about the internals of the system under test.

Others also like the name *acceptance tests* (see "On Acceptance Tests" on page 631). This distinction is less about the level of granularity of the test or the system, but more about whether the test is checking on the "acceptance criteria" for a feature (i.e., *behaviour*), as visible to the user.

Feature tests should have a human-readable story that we can follow. We make it explicit using comments that accompany the test code. When creating a new FT, we can write the comments first, to capture the key points of the user story. Being human-readable, you could even share them with nonprogrammers, as a way of discussing the requirements and features of your app.

Test-driven development and Agile or Lean software development methodologies often go together, and one of the things we tend to talk about is the minimum viable app: what is the simplest thing we can build that is still useful? Let's start by building that, so that we can test the water as quickly as possible.

A minimum viable to-do list really only needs to let the user enter some to-do items, and remember them for their next visit.

1 If you want to read more about user stories, check out Gojko Adzic's *Fifty Quick Ideas to Improve Your User Stories* or Mike Cohn's *User Stories Applied: For Agile Software Development*.

Open up *functional_tests.py* and write a story a bit like this one:

functional_tests.py (ch02l001)

```
from selenium import webdriver

browser = webdriver.Firefox()

# Edith has heard about a cool new online to-do app.
# She goes to check out its homepage
browser.get("http://localhost:8000")

# She notices the page title and header mention to-do lists
assert "To-Do" in browser.title

# She is invited to enter a to-do item straight away

# She types "Buy peacock feathers" into a text box
# (Edith's hobby is tying fly-fishing lures)

# When she hits enter, the page updates, and now the page lists
# "1: Buy peacock feathers" as an item in a to-do list

# There is still a text box inviting her to add another item.
# She enters "Use peacock feathers to make a fly" (Edith is very methodical)

# The page updates again, and now shows both items on her list

# Satisfied, she goes back to sleep

browser.quit()
```

We Have a Word for Comments…

When I first started at PythonAnywhere, I used to virtuously pepper my code with nice descriptive comments. My colleagues said to me: "Harry, we have a word for comments. We call them lies." I was shocked! I learned in school that comments are good practice?

They were exaggerating for effect. There is definitely a place for comments that add context and intention. But my colleagues were pointing out that comments aren't always as useful as you hope. For starters, it's pointless to write a comment that just repeats what you're doing with the code:

```
# increment wibble by 1
wibble += 1
```

Not only is it pointless, but there's a danger that you'll forget to update the comments when you update the code, and they end up being misleading—lies! The ideal is to strive to make your code so readable, to use such good variable names and function

names, and to structure it so well that you no longer need any comments to explain *what* the code is doing. Just a few here and there to explain *why*.

There are other places where comments are very useful. We'll see that Django uses them a lot in the files it generates for us to use as a way of suggesting helpful bits of its API.

And, of course, we use comments to explain the user story in our functional tests—by forcing us to make a coherent story out of the test, it makes sure we're always testing from the point of view of the user.

There is more fun to be had in this area, things like *Behaviour-Driven Development* (see Online Appendix: BDD) and building domain-specific languages (DSLs) for testing, but they're topics for other books.[2]

For more on comments, I recommend John Ousterhout's *A Philosophy of Software Design*, which you can get a taste of by reading his lecture notes from the chapter on comments.

You'll notice that, apart from writing the test out as comments, I've updated the `assert` to look for "To-Do" instead of Django's "Congratulations". That means we expect the test to fail now. Let's try running it.

First, start up the server:

```
$ python manage.py runserver
```

And then, in another terminal, run the tests:

```
$ python functional_tests.py
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 10, in <module>
    assert "To-Do" in browser.title
AssertionError
```

That's what we call an *expected fail*, which is actually good news—not quite as good as a test that passes, but at least it's failing for the right reason; we can have some confidence we've written the test correctly.

2 Check out this video by the great Dave Farley if you want a taste: *https://oreil.ly/bbawE*.

The Python Standard Library's unittest Module

There are a couple of little annoyances we should probably deal with. Firstly, the message "AssertionError" isn't very helpful—it would be nice if the test told us what it actually found as the browser title. Also, it's left a Firefox window hanging around the desktop, so it would be nice if that got cleared up for us automatically.

One option would be to use the second parameter of the `assert` keyword, something like:

```
assert "To-Do" in browser.title, f"Browser title was {browser.title}"
```

And we could also use `try/finally` to clean up the old Firefox window.

But these sorts of problems are quite common in testing, and there are some ready-made solutions for us in the standard library's `unittest` module. Let's use that! In *functional_tests.py*:

functional_tests.py (ch02l003)

```
import unittest
from selenium import webdriver


class NewVisitorTest(unittest.TestCase):  ❶
    def setUp(self):  ❸
        self.browser = webdriver.Firefox()  ❹

    def tearDown(self):  ❸
        self.browser.quit()

    def test_can_start_a_todo_list(self):  ❷
        # Edith has heard about a cool new online to-do app.
        # She goes to check out its homepage
        self.browser.get("http://localhost:8000")  ❹

        # She notices the page title and header mention to-do lists
        self.assertIn("To-Do", self.browser.title)  ❺

        # She is invited to enter a to-do item straight away
        self.fail("Finish the test!")  ❻

        [...]

        # Satisfied, she goes back to sleep


if __name__ == "__main__":  ❼
    unittest.main()  ❼
```

You'll probably notice a few things here:

❶ Tests are organised into classes, which inherit from `unittest.TestCase`.

❷ The main body of the test is in a method called `test_can_start_a_todo_list`. Any method whose name starts with `test_` is a test method, and will be run by the test runner. You can have more than one `test_` method per class. Nice descriptive names for our test methods are a good idea too.

❸ `setUp` and `tearDown` are special methods that are run before and after each test. I'm using them to start and stop our browser. They're a bit like `try/finally`, in that `tearDown` will run even if there's an error during the test itself.[3] No more Firefox windows left lying around!

❹ `browser`, which was previously a global variable, becomes `self.browser`, an attribute of the test class. This lets us pass it between `setUp`, `tearDown`, and the test method itself.

❺ We use `self.assertIn` instead of just `assert` to make our test assertions. `unittest` provides lots of helper functions like this to make test assertions, like `assertEqual`, `assertTrue`, `assertFalse`, and so on. You can find more in the `unittest` documentation.

❻ `self.fail` just fails no matter what, producing the error message given. I'm using it as a reminder to finish the test.

❼ Finally, we have the `if __name__ == "__main__"` clause. (If you've not seen it before, that's how a Python script checks if it's been executed from the command line, rather than just imported by another script.) We call `unittest.main()`, which launches the `unittest` test runner, which will automatically find test classes and methods in the file and run them.

3 The only exception is that if you have an exception inside `setUp`, then `tearDown` doesn't run.

If you've read the Django testing documentation, you might have seen something called `LiveServerTestCase`, and are wondering whether we should use it now. Full points to you for reading the friendly manual! `LiveServerTestCase` is a bit too complicated for now, but I promise I'll use it in a later chapter.

Let's try out our new and improved FT![4]

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_todo_list
(__main__.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 18, in
test_can_start_a_todo_list
    self.assertIn("To-Do", self.browser.title)
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'To-Do' not found in 'The install worked successfully!
Congratulations!'

 ----------------------------------------------------------------------
Ran 1 test in 1.747s

FAILED (failures=1)
```

That's a bit nicer, isn't it? It tidied up our Firefox window, it gives us a nicely formatted report of how many tests were run and how many failed, and the `assertIn` has given us a helpful error message with useful debugging info. Bonzer!

4 Are you unable to move on because you're wondering what those *ch02l00x* things are, next to some of the code listings? They refer to specific commits in the book's example repo. It's all to do with my book's own tests. You know, the tests for the tests in the book about testing. They have tests of their own, naturally.

If you see some error messages saying `ResourceWarning` about "unclosed files", it's safe to ignore those. They seem to come and go, every few Selenium releases. They don't affect the important things to look for in our tracebacks and test results.

pytest Versus unittest

The Python world is increasingly turning from the standard-library provided `unittest` module towards a third-party tool called `pytest`. I'm a big fan too!

The Django project has a bunch of helpful tools designed to work with `unittest`. Although it is possible to get them to work with `pytest`, it felt like one thing too many to include in this book.

Read Brian Okken's Python Testing with pytest for an excellent, comprehensive guide to Pytest instead.

Commit

This is a good point to do a commit; it's a nicely self-contained change. We've expanded our functional test to include comments that describe the task we're setting ourselves, our minimum viable to-do list. We've also rewritten it to use the Python `unittest` module and its various testing helper functions.

Do a **`git status`**—that should assure you that the only file that has changed is *functional_tests.py*. Then do a **`git diff -w`**, which shows you the difference between the last commit and what's currently on disk, with the `-w` saying "ignore whitespace changes".

That should tell you that *functional_tests.py* has changed quite substantially:

```
$ git diff -w
diff --git a/functional_tests.py b/functional_tests.py
index d333591..b0f22dc 100644
--- a/functional_tests.py
+++ b/functional_tests.py
@@ -1,15 +1,24 @@
+import unittest
 from selenium import webdriver

-browser = webdriver.Firefox()

+class NewVisitorTest(unittest.TestCase):
+    def setUp(self):
+        self.browser = webdriver.Firefox()
+
+    def tearDown(self):
+        self.browser.quit()
+
+    def test_can_start_a_todo_list(self):
         # Edith has heard about a cool new online to-do app.
         # She goes to check out its homepage
-browser.get("http://localhost:8000")
+        self.browser.get("http://localhost:8000")

         # She notices the page title and header mention to-do lists
-assert "To-Do" in browser.title
+        self.assertIn("To-Do", self.browser.title)

         # She is invited to enter a to-do item straight away
+        self.fail("Finish the test!")

[...]
```

Now let's do a:

```
$ git commit -a
```

The `-a` means "automatically add any changes to tracked files" (i.e., any files that we've committed before). It won't add any brand new files (you have to explicitly `git add` them yourself), but often, as in this case, there aren't any new files, so it's a useful shortcut.

When the editor pops up, add a descriptive commit message, like "First FT specced out in comments, and now uses unittest".

Now that our FT uses a real test framework, and that we've got placeholder comments for what we want it to do, we're in an excellent position to start writing some real code for our lists app. Read on!

Useful TDD Concepts

User story
: A description of how the application will work from the point of view of the user; used to structure a functional test

Expected failure
: When a test fails in the way that we expected it to

CHAPTER 3

Testing a Simple Home Page with Unit Tests

We finished the last chapter with a functional test (FT) failing, telling us that it wanted the home page for our site to have "To-Do" in its title. Time to start working on our application. In this chapter, we'll build our first HTML page, find out about URL handling, and create responses to HTTP requests with Django's view functions.

Warning: Things Are About to Get Real

The first two chapters were intentionally nice and light. From now on, we get into some more meaty coding. Here's a prediction: at some point, things are going to go wrong. You're going to see different results from what I say you should see. This is a Good Thing, because it will be a genuine character-building Learning Experience™.

One possibility is that I've given some ambiguous explanations, and you've done something different from what I intended. Step back and have a think about what we're trying to achieve at this point in the book. Which file are we editing, what do we want the user to be able to do, what are we testing and why? It may be that you've edited the wrong file or function, or are running the wrong tests. I reckon you'll learn more about TDD from these "stop and think" moments than you do from all the times when following instructions and copy-pasting goes smoothly.

Or it may be a real bug. Be tenacious, read the error message carefully (see "Reading Tracebacks" on page 38), and you'll get to the bottom of it. It's probably just a missing comma, or trailing slash, or a missing *s* in one of the Selenium find methods. But, as Zed Shaw memorably insisted in *Learn Python The Hard Way*, debugging is also an absolutely vital part of learning, so do stick it out! You can always drop me an *email* if you get really stuck. Happy debugging!

Our First Django App and Our First Unit Test

Django encourages you to structure your code into *apps*. The theory is that one project can have many apps; you can use third-party apps developed by other people, and you might even reuse one of your own apps in a different project…although I have to say, I've never actually managed the latter, myself! Still, apps are a good way to keep your code organised.

Let's start an app for our to-do lists:

```
$ python manage.py startapp lists
```

That will create a folder called *lists*, next to *manage.py* and the existing *superlists* folder, and within it a number of placeholder files for things like models, views, and, of immediate interest to us, tests:

```
.
├── db.sqlite3
├── functional_tests.py
├── lists
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Unit Tests, and How They Differ from Functional Tests

As with so many of the labels we put on things, the line between unit tests and FTs can become a little blurry at times. The basic distinction, though, is that FTs test the application from the outside, from the user's point of view. Unit tests on the other hand test the application from the inside, from the programmer's point of view.

The TDD approach I'm demonstrating uses both types of test to drive the development of our application, and ensure its correctness. Our workflow will look a bit like this:

1. We start by writing a *functional test*, describing a typical example of our new functionality from the user's point of view.
2. Once we have an FT that fails, we start to think about how to write code that can get it to pass (or at least to get past its current failure). We now use one or more *unit tests* to define how we want our code to behave—the idea is that each line of production code we write should be tested by (at least) one of our unit tests.
3. Once we have a failing unit test, we write the smallest amount of *application code* we can—just enough to get the unit test to pass. We may iterate between steps 2 and 3 a few times, until we think the FT will get a little further.
4. Now we can rerun our FTs and see if they pass, or get a little further. That may prompt us to write some new unit tests, and some new code, and so on.
5. Once we're comfortable that the core functionality works end-to-end, we can extend out to cover more permutations and edge cases, using just unit tests now.

You can see that, all the way through, the FTs are driving what development we do from a high level, while the unit tests drive what we do at a low level.

The FTs don't aim to cover every single tiny detail of our app's behaviour; they are there to reassure us that everything is wired up correctly. The unit tests are there to exhaustively check all the lower-level details and corner cases. See Table 3-1.

Table 3-1. Functional tests versus unit tests

Functional tests	Unit tests
One test per feature/user story	Many tests per feature
Tests from the user's point of view	Tests the code (i.e., the programmer's point of view)
Can test that the UI "really" works	Tests the internals—individual functions or classes
Provides confidence that everything is wired together correctly and works end-to-end	Can exhaustively check permutations, details, and edge cases
Can warn about problems without telling you exactly what's wrong	Can point at exactly where the problem is
Slow	Fast

Functional tests should help you build an application that actually works, and guarantee you never accidentally break it. Unit tests should help you to write code that's clean and bug free.

Enough theory for now—let's see how it looks in practice.

Unit Testing in Django

Let's see how to write a unit test for our home page view. Open up the new file at *lists/tests.py*, and you'll see something like this:

lists/tests.py

```
from django.test import TestCase

# Create your tests here.
```

Django has helpfully suggested we use a special version of `TestCase`, which it provides. It's an augmented version of the standard `unittest.TestCase`, with some additional Django-specific features, which we'll discover over the next few chapters.

You've already seen that the TDD cycle involves starting with a test that fails, then writing code to get it to pass. Well, before we can even get that far, we want to know that the unit test we're writing will definitely be run by our automated test runner, whatever it is. In the case of *functional_tests.py* we're running it directly, but this file made by Django is a bit more like magic. So, just to make sure, let's make a deliberately silly failing test:

lists/tests.py (ch03l002)

```
from django.test import TestCase


class SmokeTest(TestCase):
    def test_bad_maths(self):
        self.assertEqual(1 + 1, 3)
```

Now, let's invoke this mysterious Django test runner. As usual, it's a *manage.py* command:

```
$ python manage.py test
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
F
======================================================================
FAIL: test_bad_maths (lists.tests.SmokeTest.test_bad_maths)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 6, in test_bad_maths
    self.assertEqual(1 + 1, 3)
    ~~~~~~~~~~~~~~~~^^^^^^^^^^
AssertionError: 2 != 3

 ----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Excellent. The machinery seems to be working. This is a good point for a commit:

```
$ git status  # should show you lists/ is untracked
$ git add lists
$ git diff --staged  # will show you the diff that you're about to commit
$ git commit -m "Add app for lists, with deliberately failing unit test"
```

As you've no doubt guessed, the `-m` flag lets you pass in a commit message at the command line, so you don't need to use an editor. It's up to you to pick the way you like to use the Git command line; I'll just show you the main ones I've seen used. For me, the key rule of VCS hygiene is: *make sure you always review what you're about to commit before you do it*.

Django's MVC, URLs, and View Functions

Django is structured along a classic *model-view-controller* (MVC) pattern—well, *broadly*. It definitely does have models, but what Django calls "views" are really controllers, and the view part is actually provided by the templates, but you can see the general idea is there!

If you're interested, you can look up the finer points of the discussion in the Django FAQs.

Irrespective of any of that, as with any web server, Django's main job is to decide what to do when a user asks for a particular URL on our site. Django's workflow goes something like this:

1. An HTTP *request* comes in for a particular URL.
2. Django uses some rules to decide which *view* function should deal with the request (this is referred to as *resolving* the URL).
3. The view function processes the request and returns an HTTP *response*.

So, we want to test two things:

1. Can we make this view function return the HTML we need?
2. Can we tell Django to use this view function when we make a request for the root of the site ("/")?

Let's start with the first.

Unit Testing a View

Open up *lists/tests.py*, and change our silly test to something like this:

lists/tests.py (ch03l003)

```
from django.test import TestCase
from django.http import HttpRequest  ❶
from lists.views import home_page


class HomePageTest(TestCase):
    def test_home_page_returns_correct_html(self):
        request = HttpRequest()  ❶
        response = home_page(request)  ❷
        html = response.content.decode("utf8")  ❸
        self.assertIn("<title>To-Do lists</title>", html)  ❹
        self.assertTrue(html.startswith("<html>"))  ❺
        self.assertTrue(html.endswith("</html>"))  ❺
```

What's going on in this new test? Well, remember, a view function takes an HTTP request as input, and produces an HTTP response. So, to test that:

❶ We import the `HttpRequest` class so that we can then create a request object within our test. This is the kind of object that Django will create when a user's browser asks for a page.

❷ We pass the `HttpRequest` object to our `home_page` view, which gives us a response. You won't be surprised to hear that the response is an instance of a class called `HttpResponse`.

❸ Then, we extract the `.content` of the response. These are the raw bytes, the ones and zeros that would be sent down the wire to the user's browser. We call `.decode()` to convert them into the string of HTML that's being sent to the user.

❹ Now we can make some assertions: we know we want an HTML `<title>` tag somewhere in there, with the words "To-Do lists" in it—because that's what we specified in our FT.

❺ And we can do a vague sense-check that it's valid HTML by checking that it starts with an `<html>` tag, which gets closed at the end.

So, what do you think will happen when we run the tests?

```
$ python manage.py test
Found 1 test(s).
System check identified no issues (0 silenced).
E
======================================================================
ERROR: lists.tests (unittest.loader._FailedTest.lists.tests)
 ----------------------------------------------------------------------
ImportError: Failed to import test module: lists.tests
Traceback (most recent call last):
[...]
  File "...goat-book/lists/tests.py", line 3, in <module>
    from lists.views import home_page
ImportError: cannot import name 'home_page' from 'lists.views'
```

It's a very predictable and uninteresting error: we tried to import something we haven't even written yet. But it's still good news—for the purposes of TDD, an exception that was predicted counts as an expected failure. Because we have both a failing FT and a failing unit test, we have the Testing Goat's full blessing to code away.

At Last! We Actually Write Some Application Code!

It is exciting, isn't it? Be warned, TDD means that long periods of anticipation are only defused very gradually, and by tiny increments. Especially as we're learning and only just starting out, we only allow ourselves to change (or add) one line of code at a time—and each time, we make just the minimal change required to address the current test failure.

I'm being deliberately extreme here, but what's our current test failure? We can't import `home_page` from `lists.views`? OK, let's fix that—and only that. In *lists/views.py*:

lists/views.py (ch03l004)

```
from django.shortcuts import render

# Create your views here.
home_page = None
```

"You must be joking!" I can hear you say.

I can hear you because it's what I used to say (with feeling) when my colleagues first demonstrated TDD to me. Well, bear with me, and we'll talk about whether or not this is all taking it too far in a little while. But for now, let yourself follow along, even if it's with some exasperation, and see if our tests can help us write the correct code, one tiny step at a time.

Let's run the tests again:

```
[...]
  File "...goat-book/lists/tests.py", line 9, in
test_home_page_returns_correct_html
    response = home_page(request)
TypeError: 'NoneType' object is not callable
```

We still get an error, but it's moved on a bit. Instead of an import error, our tests are telling us that our `home_page` "function" is not callable. That gives us a justification for changing it from being `None` to being an actual function. At the very smallest level of detail, every single code change can be driven by the tests!

Back in *lists/views.py*:

lists/views.py (ch03l005)

```
from django.shortcuts import render


def home_page():
    pass
```

Again, we're making the smallest, simplest change we can possibly make, that addresses precisely the current test failure. Our tests wanted something callable, so we gave them the simplest possible callable thing: a function that takes no arguments and returns nothing.

Let's run the tests again and see what they think:

```
    response = home_page(request)
TypeError: home_page() takes 0 positional arguments but 1 was given
```

Once more, our error message has changed slightly, and is guiding us towards fixing the next thing that's wrong.

The Unit-Test/Code Cycle

We can start to settle into the TDD *unit-test/code cycle* now:

1. In the terminal, run the unit tests and see how they fail.
2. In the editor, make a minimal code change to address the current test failure.

And repeat!

The more nervous we are about getting our code right, the smaller and more minimal we make each code change—the idea is to be absolutely sure that each bit of code is justified by a test.

This may seem laborious—and at first, it will be. But once you get into the swing of things, you'll find yourself coding quickly even if you take microscopic steps—this is how we write all of our production code at work.

Let's see how fast we can get this cycle going:

Minimal code change:

lists/views.py (ch03l006)

```
def home_page(request):
    pass
```

Tests:

```
    html = response.content.decode("utf8")
           ^^^^^^^^^^^^^^^^
AttributeError: 'NoneType' object has no attribute 'content'
```

Code—we use `django.http.HttpResponse`, as predicted:

lists/views.py (ch03l007)

```
from django.http import HttpResponse


def home_page(request):
    return HttpResponse()
```

Tests again:

```
AssertionError: '<title>To-Do lists</title>' not found in ''
```

Code again:

lists/views.py (ch03l008)

```
def home_page(request):
    return HttpResponse("<title>To-Do lists</title>")
```

Tests yet again:

```
        self.assertTrue(html.startswith("<html>"))
AssertionError: False is not true
```

Code yet again:

lists/views.py (ch03l009)

```
def home_page(request):
    return HttpResponse("<html><title>To-Do lists</title>")
```

Tests—almost there?

```
        self.assertTrue(html.endswith("</html>"))
AssertionError: False is not true
```

Come on, one last effort:

lists/views.py (ch03l010)

```
def home_page(request):
    return HttpResponse("<html><title>To-Do lists</title></html>")
```

Surely?

```
$ python manage.py test
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
.
 ----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
Destroying test database for alias 'default'...
```

Hooray! Our first ever unit test pass! That's so momentous that I think it's worthy of a commit:

```
$ git diff  # should show changes to tests.py, and views.py
$ git commit -am "First unit test and view function"
```

That was the last variation on `git commit` I'll show, the `a` and `m` flags together, which adds all changes to tracked files and uses the commit message from the command line.[1]

`git commit -am` is the quickest formulation, but also gives you the least feedback about what's being committed, so make sure you've done a `git status` and a `git diff` beforehand, and are clear on what changes are about to go in.

1 I'm quite casual about my commit messages in this book, but in professional organisations or open source projects, people often want to be a bit more formal. Check out *https://cbea.ms/git-commit* and *https://www.conventionalcommits.org*.

Our Functional Tests Tell Us We're Not Quite Done Yet

We've got our unit test passing, so let's go back to running our FTs to see if we've made progress. Don't forget to spin up the dev server again, if it's not still running.

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_todo_list
(__main__.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 18, in
test_can_start_a_todo_list
    self.assertIn("To-Do", self.browser.title)
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'To-Do' not found in 'The install worked successfully!
Congratulations!'

 ----------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

Looks like something isn't quite right. This is the reason we have functional tests!

Do you remember at the beginning of the chapter, we said we needed to do two things: firstly, create a view function to produce responses for requests, and secondly, tell the server which functions should respond to which URLs? Thanks to our FT, we have been reminded that we still need to do the second thing.

How can we write a test for URL resolution? At the moment, we just test the view function directly by importing it and calling it. But we want to test more layers of the Django stack. Django, like most web frameworks, supplies a tool for doing just that, called the Django test client.

Let's see how to use it by adding a second, alternative test to our unit tests:

lists/tests.py (ch03l011)

```
class HomePageTest(TestCase):
    def test_home_page_returns_correct_html(self):  ❶
        request = HttpRequest()
        response = home_page(request)
        html = response.content.decode("utf8")
        self.assertIn("<title>To-Do lists</title>", html)
        self.assertTrue(html.startswith("<html>"))
        self.assertTrue(html.endswith("</html>"))

    def test_home_page_returns_correct_html_2(self):
        response = self.client.get("/")  ❷
        self.assertContains(response, "<title>To-Do lists</title>")  ❸
```

❶ This is our existing test.

❷ In our new test, we access the test client via `self.client`, which is available on any test that uses `django.test.TestCase`. It provides methods like `.get()`, which simulates a browser making HTTP requests, and takes a URL as its first parameter. We use this instead of manually creating a request object and calling the view function directly.

❸ Django also provides some assertion helpers like `assertContains`, which save us from having to manually extract and decode response content, and have some other nice properties besides, as we'll see.

Let's see how that works:

```
$ python manage.py test
Found 2 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.F
======================================================================
FAIL: test_home_page_returns_correct_html_2
(lists.tests.HomePageTest.test_home_page_returns_correct_html_2)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 17, in
test_home_page_returns_correct_html_2
    self.assertContains(response, "<title>To-Do lists</title>")
[...]
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)

 ----------------------------------------------------------------------
Ran 2 tests in 0.004s

FAILED (failures=1)
Destroying test database for alias 'default'...
```

Hmm, something about 404s? Let's dig into it.

Reading Tracebacks

Let's spend a moment talking about how to read tracebacks, as it's something we have to do a lot in TDD. You soon learn to scan through them and pick up relevant clues:

```
======================================================================
FAIL: test_home_page_returns_correct_html_2  ❷
(lists.tests.HomePageTest.test_home_page_returns_correct_html_2)  ❷
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 17, in
test_home_page_returns_correct_html_2
    self.assertContains(response, "<title>To-Do lists</title>")  ❸
    ~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  ❹
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404  ❶
(expected 200)

 ----------------------------------------------------------------------
[...]
```

❶ The first place you look is usually *the error itself*. Sometimes that's all you need to see, and it will let you identify the problem immediately. But sometimes, like in this case, it's not quite self-evident.

❷ The next thing to double-check is: *which test is failing?* Is it definitely the one we expected—that is, the one we just wrote? In this case, the answer is yes.

❸ Then we look for the place in *our test code* that kicked off the failure. We work our way down from the top of the traceback, looking for the filename of the tests file to check which test function, and what line of code, the failure is coming from. In this case, it's the line where we call the `assertContains` method.

❹ In Python 3.11 and later, you can also look out for the string of carets, which try to tell you exactly where the exception came from. This is more useful for unexpected exceptions than for assertion failures like we have now.

There is ordinarily a fifth step, where we look further down for any of *our own application code* that was involved with the problem. In this case, it's all Django code, but we'll see plenty of examples of this fifth step later in the book.

Pulling it all together, we interpret the traceback as telling us that:

- When we tried to do our assertion on the content of the response.
- Django's test helpers failed, saying that they could not do that.
- Because the response is an HTML 404 Not Found error, instead of a normal 200 OK response.

In other words, Django isn't yet configured to respond to requests for the root URL ("/") of our site. Let's make that happen now.

urls.py

Django uses a file called *urls.py* to map URLs to view functions. This mapping is also called *routing*. There's a main *urls.py* for the whole site in the *superlists* folder. Let's go take a look:

superlists/urls.py

```
"""
URL configuration for superlists project.

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/5.2/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""

from django.contrib import admin
from django.urls import path

urlpatterns = [
    path("admin/", admin.site.urls),
]
```

As usual, lots of helpful comments and default suggestions from Django. In fact, that very first example is pretty much exactly what we want! Let's use that, with some minor changes:

superlists/urls.py (ch03l012)

```
from django.urls import path  ❶
from lists.views import home_page  ❷

urlpatterns = [
    path("", home_page, name="home"),  ❸
]
```

❶ No need to import `admin` from `django.contrib`. Django's admin site is amazing, but it's a topic for another book.

❷ But we will import our home page view function.

❸ And we wire it up here, as a `path()` entry in the `urlpatterns` global. Django strips the leading slash from all URLs, so `"/url/path/to"` becomes `"url/path/to"` and the base URL is just the empty string, `""`. So this config says, the "base URL should point to our home page view".

Now we can run our unit tests again, with **`python manage.py test`**:

```
[...]
..
 ----------------------------------------------------------------------
Ran 2 tests in 0.003s

OK
```

Hooray!

Time for a little tidy-up. We don't need two separate tests, so let's move everything out of our low-level test that calls the view function directly, into the test that uses the Django test client:

lists/tests.py (ch03l013)

```
class HomePageTest(TestCase):
    def test_home_page_returns_correct_html(self):
        response = self.client.get("/")
        self.assertContains(response, "<title>To-Do lists</title>")
        self.assertContains(response, "<html>")
        self.assertContains(response, "</html>")
```

Why Didn't We Just Use the Django Test Client All Along?

You may be asking yourself, "Why didn't we just use the Django test client from the very beginning?" In real life, that's what I would do. But I wanted to show you the "manual" way of doing it first, for a couple of reasons. Firstly, because it enabled me to introduce concepts one by one, and keep the learning curve as shallow as possible. Secondly, because you may not always be using Django to build your apps, and testing tools may not always be available—but calling functions directly and examining their responses is always possible!

The Django test client does also have disadvantages; later in the book (in Chapter 27) we'll discuss the difference between fully isolated unit tests and the types of test that the test client pushes us towards (people often say these are technically "integration tests"). But for now, it's very much the pragmatic choice.

But now the moment of truth: will our functional tests pass?

```
$ python functional_tests.py
[...]
======================================================================
FAIL: test_can_start_a_todo_list
(__main__.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 21, in
test_can_start_a_todo_list
    self.fail("Finish the test!")
AssertionError: Finish the test!
```

Failed? What? Oh, it's just our little reminder? Yes? Yes! We have a web page!

Ahem. Well, *I* thought it was a thrilling end to the chapter. You may still be a little baffled, perhaps keen to hear a justification for all these tests (and don't worry; all that will come), but I hope you felt just a tinge of excitement near the end there.

Just a little commit to calm down, and reflect on what we've covered:

```
$ git diff  # should show our modified test in tests.py, and the new config in urls.py
$ git commit -am "url config, map / to home_page view"
```

That was quite a chapter! Why not try typing `git log`, possibly using the `--oneline` flag, for a reminder of what we got up to:

```
$ git log --oneline
a6e6cc9 url config, map / to home_page view
450c0f3 First unit test and view function
ea2b037 Add app for lists, with deliberately failing unit test
[...]
```

Not bad—we covered the following:

- Starting a Django app
- The Django unit test runner
- The difference between FTs and unit tests
- Django view functions, and request and response objects
- Django URL resolving and *urls.py*
- The Django test client
- Returning basic HTML from a view

Useful Commands and Concepts

Running the Django dev server

```
python manage.py runserver
```

Running the functional tests

```
python functional_tests.py
```

Running the unit tests

```
python manage.py test
```

The unit-test/code cycle

1. Run the unit tests in the terminal.
2. Make a minimal code change in the editor.
3. Repeat!

CHAPTER 4

What Are We Doing with All These Tests? (And, Refactoring)

Now that we've seen the basics of TDD in action, it's time to pause and talk about why we're doing it.

I'm imagining several of you, dear readers, have been holding back some seething frustration—perhaps some of you have done a bit of unit testing before, and perhaps some of you are just in a hurry. You've been biting back questions like:

- Aren't all these tests a bit excessive?
- Surely some of them are redundant? There's duplication between the functional tests and the unit tests.
- Those unit tests seemed way too trivial—testing a one-line function that returns a constant! Isn't that just a waste of time? Shouldn't we save our tests for more complex things?
- What about all those tiny changes during the unit-test/code cycle? Couldn't we just skip to the end? I mean, `home_page = None`!? Really?
- You're not telling me you *actually* code like this in real life?

Ah, young grasshopper. I too was once full of questions like these. But only because they're perfectly good questions. In fact, I still ask myself questions like these—all the time. Does all this stuff really have value? Is this a bit of a cargo cult?

Programming Is Like Pulling a Bucket of Water Up from a Well

Ultimately, programming is hard. Often, we are smart, so we succeed. TDD is there to help us out when we're not so smart. Kent Beck (who basically invented TDD) uses the metaphor of lifting a bucket of water out of a well with a rope: when the well isn't too deep, and the bucket isn't very full, it's easy. And even lifting a full bucket is pretty easy at first. But after a while, you're going to get tired. TDD is like having a ratchet that lets you save your progress, so you can take a break, and make sure you never slip backwards.

That way, you don't have to be smart *all* the time (see Figure 4-1).

Figure 4-1. Test ALL the things (adapted from Allie Brosh, Hyperbole and a Half)

OK, perhaps *in general*, you're prepared to concede that TDD is a good idea, but maybe you still think I'm overdoing it? Testing the tiniest thing, and taking ridiculously many small steps?

TDD is a *discipline*, and that means it's not something that comes naturally. Because many of the payoffs aren't immediate but only come in the longer term, you have to force yourself to do it in the moment. That's what the image of the Testing Goat is supposed to represent—you need to be a bit bloody-minded about it.

On the Merits of Trivial Tests for Trivial Functions

In the short term, it may feel a bit silly to write tests for simple functions and constants.

It's perfectly possible to imagine still doing "mostly" TDD, but following more relaxed rules where you don't unit test *absolutely* everything. But in this book my aim is to demonstrate full, rigorous TDD. Like a kata in a martial art, the idea is to learn the motions in a controlled context, when there is no adversity, so that the techniques are part of your muscle memory. It seems trivial now, because we've started with a very simple example. The problem comes when your application gets complex—that's when you really need your tests. And the danger is that complexity tends to sneak up on you, gradually. You may not notice it happening, but soon you're a boiled frog.

There are two other things to say in favour of tiny, simple tests for simple functions.

Firstly, if they're really trivial tests, then they won't take you that long to write. So stop moaning and just write them already.

Secondly, it's always good to have a placeholder. Having a test *there* for a simple function means it's that much less of a psychological barrier to overcome when the simple function gets a tiny bit more complex—perhaps it grows an `if`. Then a few weeks later, it grows a `for` loop. Before you know it, it's a recursive metaclass-based polymorphic tree parser factory. But because it's had tests from the very beginning, adding a new test each time has felt quite natural, and it's well tested. The alternative involves trying to decide when a function becomes "complicated enough", which is highly subjective. And worse, because there's no placeholder, it feels like that much more effort to start, so you're tempted each time to put it off…and pretty soon—frog soup!

Instead of trying to figure out some hand-wavy subjective rules for when you should write tests, and when you can get away with not bothering, I suggest following the discipline for now—and as with any discipline, you have to take the time to learn the rules before you can break them.

Now, let us return to our muttons.

Using Selenium to Test User Interactions

Where were we at the end of the last chapter? Let's rerun the test and find out:

```
$ python functional_tests.py
F
======================================================================
FAIL: test_can_start_a_todo_list
(__main__.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 21, in
test_can_start_a_todo_list
    self.fail("Finish the test!")
    ~~~~~~~~~^^^^^^^^^^^^^^^^^^^^
AssertionError: Finish the test!

 ----------------------------------------------------------------------
Ran 1 test in 1.609s

FAILED (failures=1)
```

Did you try it, and get an error saying "Problem loading page" or "Unable to connect"? So did I. It's because we forgot to spin up the dev server first using `manage.py runserver`. Do that, and you'll get the failure message we're after.

One of the great things about TDD is that you never have to worry about forgetting what to do next—just rerun your tests and they will tell you what you need to work on.

"Finish the test", it says, so let's do just that! Open up *functional_tests.py* and we'll extend our FT:

functional_tests.py (ch04l001)

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time
import unittest


class NewVisitorTest(unittest.TestCase):
    def setUp(self):
        self.browser = webdriver.Firefox()

    def tearDown(self):
        self.browser.quit()

    def test_can_start_a_todo_list(self):
        # Edith has heard about a cool new online to-do app.
        # She goes to check out its homepage
        self.browser.get("http://localhost:8000")

        # She notices the page title and header mention to-do lists
        self.assertIn("To-Do", self.browser.title)
        header_text = self.browser.find_element(By.TAG_NAME, "h1").text  ❶
        self.assertIn("To-Do", header_text)

        # She is invited to enter a to-do item straight away
        inputbox = self.browser.find_element(By.ID, "id_new_item")  ❶
        self.assertEqual(inputbox.get_attribute("placeholder"), "Enter a to-do item")

        # She types "Buy peacock feathers" into a text box
        # (Edith's hobby is tying fly-fishing lures)
        inputbox.send_keys("Buy peacock feathers")  ❷

        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)  ❸
        time.sleep(1)  ❹

        table = self.browser.find_element(By.ID, "id_list_table")
        rows = table.find_elements(By.TAG_NAME, "tr")  ❶
        self.assertTrue(any(row.text == "1: Buy peacock feathers" for row in rows))

        # There is still a text box inviting her to add another item.
        # She enters "Use peacock feathers to make a fly"
        # (Edith is very methodical)
        self.fail("Finish the test!")

        # The page updates again, and now shows both items on her list
        [...]
```

❶ We're using the two methods that Selenium provides to examine web pages: `find_element` and `find_elements` (notice the extra `s`, which means it will return several elements rather than just one). Each one is parameterised with a `By.SOMETHING`, which lets us search using different HTML properties and attributes.

❷ We also use `send_keys`, which is Selenium's way of typing into input elements.

❸ The `Keys` class (don't forget to import it) lets us send special keys like Enter.[1]

❹ When we hit Enter, the page will refresh. The `time.sleep` is there to make sure the browser has finished loading before we make any assertions about the new page. This is called an "explicit wait" (a very simple one; we'll improve it in Chapter 6).

Watch out for the difference between the Selenium `find_element()` and `find_elements()` functions. One returns an element and raises an exception if it can't find it, whereas the other returns a list, which may be empty.

Also, just look at that `any()` function. It's a little-known Python built-in. I don't even need to explain it, do I? Python is such a joy.[2]

If you're one of my readers who doesn't know Python, what's happening *inside* the `any()` may need some explaining. The basic syntax is that of a *list comprehension*, and if you haven't learned about them, you should do so immediately! Trey Hunner's explanation is excellent. In point of fact, because we're omitting the square brackets, we're actually using a *generator expression* rather than a list comprehension. It's probably less important to understand the difference between those two, but if you're curious, Guido van Rossum, the inventor of Python, has written a blog post explaining the difference.

1 You could also just use the string `"\n"`, but `Keys` also lets you send special keys like Ctrl, so I thought I'd show it.

2 Python *is* most definitely a joy, but if you think I'm being a bit smug here, I don't blame you! Actually, I wish I'd picked up on this feeling of self-satisfaction and seen it as a warning sign that I was being a little *too* clever. In the next chapter, you'll see I get my comeuppance.

Let's see how it gets on:

```
$ python functional_tests.py
[...]
  File "...goat-book/functional_tests.py", line 22, in
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: h1; For documentation on this error, please visit: [...]
```

Decoding that, the test is saying it can't find an `<h1>` element on the page. Let's see what we can do to add that to the HTML of our home page.

Big changes to a functional test are usually a good thing to commit on their own. I failed to do so when I was first working out the code for this chapter, and I regretted it later when I changed my mind and had the change mixed up with a bunch of others. The more atomic your commits, the better:

```
$ git diff  # should show changes to functional_tests.py
$ git commit -am "Functional test now checks we can input a to-do item"
```

The "Don't Test Constants" Rule, and Templates to the Rescue

Let's take a look at our unit tests, *lists/tests.py*. Currently we're looking for specific HTML strings, but that's not a particularly efficient way of testing HTML. In general, one of the rules of unit testing is "don't test constants", and testing HTML as text is a lot like testing a constant.

In other words, if you have some code that says:

```
wibble = 3
```

There's not much point in a test that says:

```
from myprogram import wibble
assert wibble == 3
```

Unit tests are really about testing logic, flow control, and configuration. Making assertions about exactly what sequence of characters we have in our HTML strings isn't doing that.

It's not *quite* that simple, because HTML is code after all, and we do want something to check that we've written code that works—but that's our FT's job, not the unit test's.

So maybe "don't test constants" isn't the online guideline at play here, but in any case, mangling raw strings in Python really isn't a great way of dealing with HTML. There's a much better solution, which is to use templates. Quite apart from anything else, if we can keep HTML to one side in a file whose name ends in *.html*, we'll get better syntax highlighting!

There are lots of Python templating frameworks out there, and Django has its own which works very well. Let's use that.

Refactoring to Use a Template

What we want to do now is make our view function return exactly the same HTML, but just using a different process. That's a refactor—when we try to improve the code *without changing its functionality*.

That last bit is really important. If you try to add new functionality at the same time as refactoring, you're much more likely to run into trouble. Refactoring is actually a whole discipline in itself, and it even has a reference book: Martin Fowler's *Refactoring*.

The first rule is that you can't refactor without tests. Thankfully, we're doing TDD, so we're way ahead of the game. Let's check that our tests pass; they will be what makes sure that our refactoring is behaviour-preserving:

```
$ python manage.py test
[...]
OK
```

Great! We'll start by taking our HTML string and putting it into its own file. Create a directory called *lists/templates* to keep templates in, and then open a file at *lists/templates/home.html*, to which we'll transfer our HTML:[3]

lists/templates/home.html (ch04l002)

```
<html>
  <title>To-Do lists</title>
</html>
```

Mmm, syntax-highlighted…much nicer! Now to change our view function:

lists/views.py (ch04l003)

```
from django.shortcuts import render


def home_page(request):
    return render(request, "home.html")
```

Instead of building our own `HttpResponse`, we now use the Django `render()` function. It takes the request as its first parameter (for reasons we'll go into later) and

3 Some people like to use another subfolder named after the app (i.e., *lists/templates/lists*) and then refer to the template as *lists/home.html*. This is called "template namespacing". I figured it was overcomplicated for this small project, but it may be worth it on larger projects. There's more in the Django tutorial.

the name of the template to render. Django will automatically search folders called *templates* inside any of your apps' directories. Then it builds an `HttpResponse` for you, based on the content of the template.

Templates are a very powerful feature of Django's, and their main strength consists of substituting Python variables into HTML text. We're not using this feature yet, but we will in future chapters. That's why we use `render()` rather than, say, manually reading the file from disk with the built-in `open()`.

Let's see if it works:

```
$ python manage.py test
[...]
======================================================================
ERROR: test_home_page_returns_correct_html
(lists.tests.HomePageTest.test_home_page_returns_correct_html)  ❷
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 7, in test_home_page_returns_correct_html
    response = self.client.get("/")  ❸
               ^^^^^^^^^^^^^^^^^^^^
[...]
  File "...goat-book/lists/views.py", line 4, in home_page
    return render(request, "home.html")  ❹
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../django/shortcuts.py", line 24, in render
    content = loader.render_to_string(template_name, context, request, using=using)
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../django/template/loader.py", line 61, in render_to_string
    template = get_template(template_name, using=using)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File ".../django/template/loader.py", line 19, in get_template
    raise TemplateDoesNotExist(template_name, chain=chain)
django.template.exceptions.TemplateDoesNotExist: home.html  ❶

----------------------------------------------------------------------
Ran 1 test in 0.074s
```

Another chance to analyse a traceback:

❶ We start with the error: it can't find the template.

❷ Then we double-check what test is failing: sure enough, it's our test of the view HTML.

❸ Then we find the line in our tests that caused the failure: it's when we request the root URL ("/").

❹ Finally, we look for the part of our own application code that caused the failure: it's when we try to call `render`.

So why can't Django find the template? It's right where it's supposed to be, in the *lists/templates* folder.

The thing is that we haven't yet *officially* registered our lists app with Django. Unfortunately, just running the `startapp` command and having what is obviously an app in your project folder isn't quite enough. You have to tell Django that you *really* mean it, and add it to *settings.py* as well—belt and braces. Open it up and look for a variable called `INSTALLED_APPS`, to which we'll add `lists`:

superlists/settings.py (ch04l004)

```
# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "lists",
]
```

You can see there's lots of apps already in there by default. We just need to add ours to the bottom of the list. Don't forget the trailing comma—it may not be required, but one day you'll be really annoyed when you forget it and Python concatenates two strings on different lines…

Now we can try running the tests again:

```
$ python manage.py test
[...]
OK
```

And we can double-check with the FTs:

```
$ python functional_tests.py
[...]
  File "...goat-book/functional_tests.py", line 22, in
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: h1; For documentation on this error, please visit: [...]
```

Good, they still get to the same place they did before. Our refactor of the code is now complete, and the tests mean we're happy that behaviour is preserved. Now we can change the tests so that they're no longer testing constants; instead, they should just check that we're rendering the right template.

Revisiting Our Unit Tests

Our unit tests are currently essentially checking HTML by hand—certainly that's very close to "testing constants".

lists/tests.py

```
def test_home_page_returns_correct_html(self):
    response = self.client.get("/")
    self.assertContains(response, "<title>To-Do lists</title>")  ❶
    self.assertContains(response, "<html>")
    self.assertContains(response, "</html>")
```

We don't want to be duplicating the full content of our HTML template in our tests, or even last sections of it. What could we do instead?

Rather than testing the full template, we could just check that we're using the *right* template. The Django test client has a method, `assertTemplateUsed`, which will let us do just that.

lists/tests.py (ch04l005)

```
def test_home_page_returns_correct_html(self):
    response = self.client.get("/")
    self.assertContains(response, "<title>To-Do lists</title>")  ❶
    self.assertContains(response, "<html>")
    self.assertContains(response, "</html>")
    self.assertTemplateUsed(response, "home.html")  ❷
```

❶ We'll leave the old tests there for now, just to make sure everything is working the way we think it is.

❷ `.assertTemplateUsed` lets us check what template was used to render a response. (NB: It will only work for responses that were retrieved by the test client.)

And that test will still pass:

```
Ran 1 tests in 0.016s

OK
```

Just because I'm always suspicious of a test I haven't seen fail, let's deliberately break it:

lists/tests.py (ch04l006)

```
self.assertTemplateUsed(response, "wrong.html")
```

That way, we'll also learn what its error messages look like:

```
AssertionError: False is not true : Template 'wrong.html' was not a template
used to render the response. Actual template(s) used: home.html
```

That's very helpful!

Let's change the assert back to the right thing.

lists/tests.py (ch04l007)

```
from django.test import TestCase


class HomePageTest(TestCase):
    def test_uses_home_template(self):
        response = self.client.get("/")
        self.assertTemplateUsed(response, "home.html")
```

Now, instead of testing constants we're testing at a higher level of abstraction. Great!

Test Behaviour, Not Implementation

As so often in the world of programming though, things are not black and white.

Yes, on the plus side, our tests no longer care about the specific content of our HTML so they are no longer brittle with respect to minor changes of the copy in our template.

But on the other hand, they depend on some Django implementation details, so they *are* brittle with respect to changing the template rendering library, or even just renaming templates.

In a way, testing for the template name (and implicitly, even checking that we used a template at all) is a lot like testing implementation. So what is the *behaviour* that we want?

Yes, in a sense, the "behaviour" we want from the view is "render the template". But from the point of view of the user, it's "show me the home page".

We're also vulnerable to accidentally breaking the template. Let's try it now, by just deleting all the contents of the template file:

```
$ mv lists/templates/home.html lists/templates/home.html.bak
$ touch lists/templates/home.html
$ python manage.py test
[...]
OK
```

Yes, our FTs will pick up on this, so ultimately we're OK:

```
$ python functional_tests.py
[...]
    self.assertIn("To-Do", self.browser.title)
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'To-Do' not found in ''
```

But it would be nice to have our unit tests pick up on this too:

```
$ mv lists/templates/home.html.bak lists/templates/home.html
```

Deciding exactly what to test with FTs and what to test with unit tests is a fine line, and the objective is not to double-test everything. But in general, the more we can test with unit tests the better. They run faster, and they give more specific feedback.

So, let's bring back a minimal "smoke test"[4] to check that what we're rendering is actually the home page:

lists/tests.py (ch04l008)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        response = self.client.get("/")
        self.assertTemplateUsed(response, "home.html")  ❶

    def test_renders_homepage_content(self):
        response = self.client.get("/")
        self.assertContains(response, "To-Do")  ❷
```

❶ We'll keep this first test, which asserts on whether we're rendering the right "constant".

❷ And this gives us a minimal smoke test that we have got the right content in the template.

4 A smoke test is a minimal test that can quickly tell you if something is wrong, without exhaustively testing every aspect that you might care about. Wikipedia has some fun speculation on the etymology.

As our home page template gains more functionality over the next couple of chapters, we'll come back to talking about what to test here in the unit tests and what to leave to the FTs.

Unit tests give you faster and more specific feedback than FTs. Bear this in mind when deciding what to test where.

We'll visit the trade-offs between different types of tests at several points in the book, and particularly in Chapter 27.

On Refactoring

That was an absolutely trivial example of refactoring. But, as Kent Beck puts it in *Test-Driven Development: By Example*, "Am I recommending that you actually work this way? No. I'm recommending that you be *able* to work this way".

In fact, as I was writing this my first instinct was to dive in and change the test first—make it use the `assertTemplateUsed()` function straight away; against the expected render; and then go ahead and make the code change. But notice how that actually would have left space for me to break things: I could have defined the template as containing *any* arbitrary string, instead of the string with the right `<html>` and `<title>` tags.

When refactoring, work on either the code or the tests, but not both at once.

There's always a tendency to skip ahead a couple of steps, to make a few tweaks to the behaviour while you're refactoring. But pretty soon you've got changes to half a dozen different files, you've totally lost track of where you are, and nothing works anymore. If you don't want to end up like Refactoring Cat (Figure 4-2), stick to small steps; keep refactoring and functionality changes entirely separate.

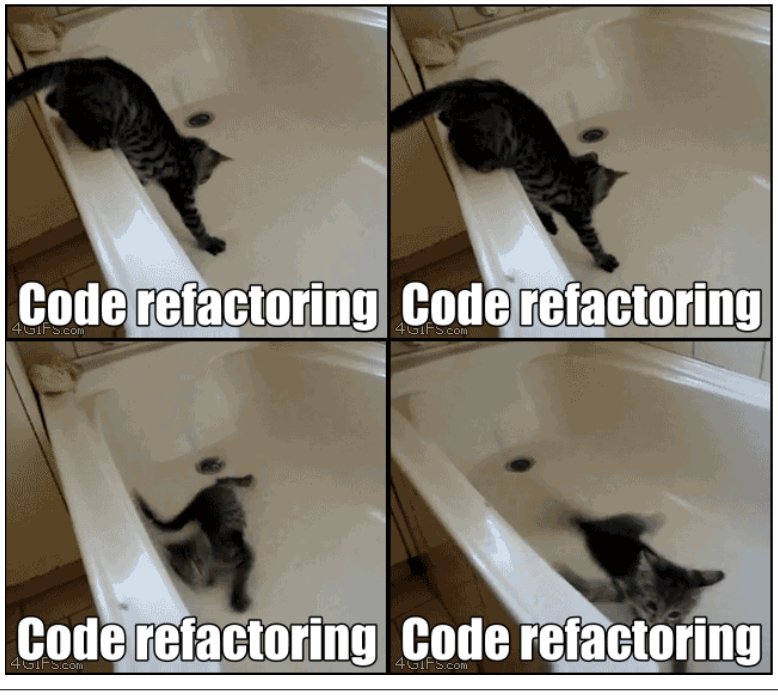


Figure 4-2. Refactoring Cat—be sure to look up the full animated GIF (source: 4GIFs.com)

We'll come across Refactoring Cat again during this book, as an example of what happens when we get carried away and change too many things at once. Think of it as the little cartoon demon counterpart to the Testing Goat, popping up over your other shoulder and giving you bad advice.

It's a good idea to do a commit after any refactoring:

```
$ git status # see tests.py, views.py, settings.py, + new templates folder
$ git add .  # will also add the untracked templates folder
$ git diff --staged # review the changes we're about to commit
$ git commit -m "Refactor homepage view to use a template"
```

A Little More of Our Front Page

In the meantime, our FT is still failing. Let's now make an actual code change to get it passing. Because our HTML is now in a template, we can feel free to make changes to it, without needing to write any extra unit tests.

This is another distinction between FTs and unit tests; because the FTs use a real web browser, we use them as the primary tool for testing our UI, and the HTML that implements it.

So, we wanted an `<h1>`:

lists/templates/home.html (ch04l009)

```
<html>
  <head>
    <title>To-Do lists</title>
  </head>
  <body>
    <h1>Your To-Do list</h1>
  </body>
</html>
```

Let's see if our FT likes it a little better:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]; For documentation on this error, [...]
```

OK, let's add an input with that ID:

lists/templates/home.html (ch04l010)

```
  [...]
  <body>
    <h1>Your To-Do list</h1>
    <input id="id_new_item" />
  </body>
</html>
```

And now what does the FT say?

```
AssertionError: '' != 'Enter a to-do item'
```

We add our placeholder text…

lists/templates/home.html (ch04l011)

```
    <input id="id_new_item" placeholder="Enter a to-do item" />
```

Which gives:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
```

So we can go ahead and put the table onto the page. At this stage it'll just be empty:

lists/templates/home.html (ch04l012)

```
    <input id="id_new_item" placeholder="Enter a to-do item" />
    <table id="id_list_table">
    </table>
  </body>
```

What does the FT think?

```
[...]
  File "...goat-book/functional_tests.py", line 40, in
test_can_start_a_todo_list
    self.assertTrue(any(row.text == "1: Buy peacock feathers" for row in rows))
    ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: False is not true
```

Slightly cryptic! We can use the line number to track it down, and it turns out it's that `any()` function I was so smug about earlier—or, more precisely, the `assertTrue`, which doesn't have a very explicit failure message. We can pass a custom error message as an argument to most `assertX` methods in `unittest`:

functional_tests.py (ch04l013)

```
        self.assertTrue(
            any(row.text == "1: Buy peacock feathers" for row in rows),
            "New to-do item did not appear in table",
        )
```

If you run the FT again, you should see our helpful message:

```
AssertionError: False is not true : New to-do item did not appear in table
```

But now, to get this to pass, we will need to actually process the user's form submission. And that's a topic for the next chapter.

For now let's do a commit:

```
$ git diff
$ git commit -am "Front page HTML now generated from a template"
```

Thanks to a bit of refactoring, we've got our view set up to render a template, we've stopped testing constants, and we're now well placed to start processing user input.

Recap: The TDD Process

We've now seen all the main aspects of the TDD process, in practice:

- Functional tests
- Unit tests
- The unit-test/code cycle
- Refactoring

It's time for a little recap, and perhaps even some flowcharts. (Forgive me, my years misspent as a management consultant have ruined me. On the plus side, said flowcharts will feature recursion!)

What does the overall TDD process look like?

- We write a test.
- We run the test and see it fail.
- We write some minimal code to get it a little further.
- We rerun the test and repeat until it passes (the unit-test/code cycle)
- Then, we look for opportunities to refactor our code, using our tests to make sure we don't break anything.
- Then, we look for opportunities to refactor our tests too, while attempting to stick to rules like "test behaviour, not implementation" and "don't test constants".
- And start again from the top!

See Figure 4-3.

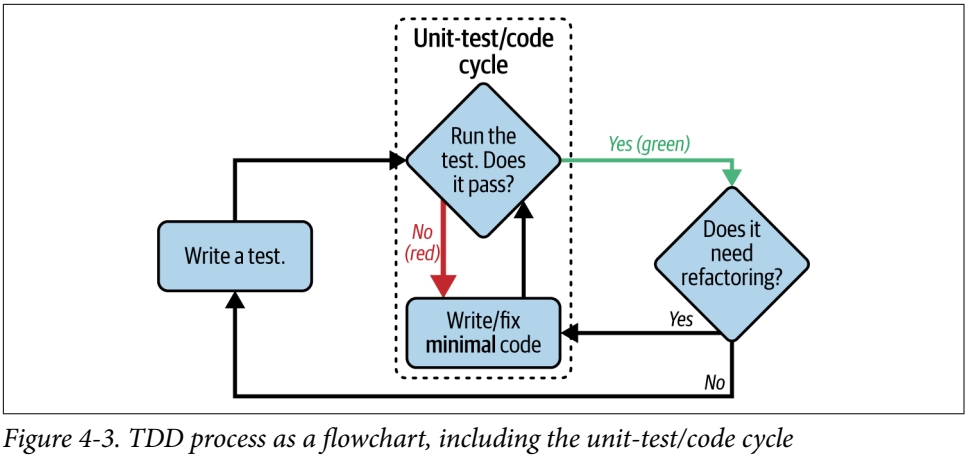


Figure 4-3. TDD process as a flowchart, including the unit-test/code cycle

It's very common to talk about this process using the three words: *red/green/refactor*. See Figure 4-4.

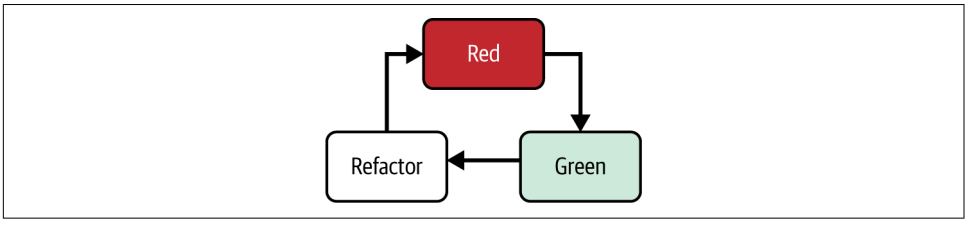


Figure 4-4. Red/green/refactor

- We write a test, and see it fail ("red").
- We cycle between code and tests until the test passes ("green").
- Then, we look for opportunities to refactor.
- Repeat as required!

Double-Loop TDD

But how does this apply when we have functional tests *and* unit tests? Well, you can think of the FT as driving a higher-level version of the same cycle, with an inner red/green/refactor loop being required to get an FT from red to green; see Figure 4-5.

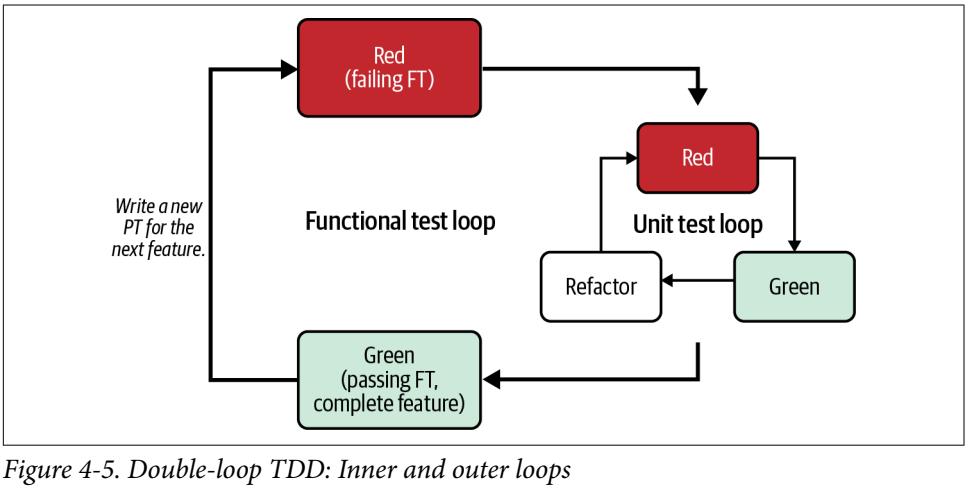


Figure 4-5. Double-loop TDD: Inner and outer loops

When a new feature or business requirement comes along, we write a new (failing) FT to capture a high-level view of the requirement. It may not cover every last edge case, but it should be enough to reassure ourselves that things are working.

To get that FT to green, we then enter into the lower-level unit test cycle, where we put together all the moving parts required, and add tests for all the edge cases. Any time we get to green and refactored at the unit test level, we can pop back up to the FT level to guide us towards the next thing we need to work on. Once both levels are green, we can do any extra refactoring or work on edge cases.

We'll explore all of the different parts of this workflow in more detail over the coming chapters.

How to "Check" Your Code, or Skip Ahead (If You Must)

All of the code examples I've used in the book are available in my repo on GitHub. So, if you ever want to compare your code against mine, you can take a look at it there.

Each chapter has its own branch, which is named after its short name. The one for this chapter is a snapshot of the code as it should be at the *end* of the chapter.

You can find a full list of them in Appendix C, as well as instructions on how to download them or use Git to compare your code to mine.

Obviously I can't possibly condone it, but you can also use my repo to "skip ahead" and check out the code to let you work on a later chapter without having worked through all the earlier chapters yourself. You're only cheating yourself you know!

CHAPTER 5
Saving User Input: Testing the Database

So far, we've managed to return a static HTML page with an input box in it. Next, we want to take the text that the user types into that input box and send it to the server, so that we can save it somehow and display it back to them later.

The first time I started writing code for this chapter, I immediately wanted to skip to what I thought was the right design: multiple database tables for lists and list items, a bunch of different URLs for adding new lists and items, three new view functions, and about half a dozen new unit tests for all of the above. But I stopped myself. Although I was pretty sure I was smart enough to handle coding all those problems at once, the point of TDD is to enable you to do one thing at a time, when you need to. So I decided to be deliberately short-sighted, and at any given moment *only* do what was necessary to get the functional tests (FTs) a little further.

This will be a demonstration of how TDD can support an incremental, iterative style of development—it may not be the quickest route, but you do get there in the end.[1] There's a neat side benefit, which is that it enables me to introduce new concepts like models, dealing with POST requests, Django template tags, and so on, *one at a time* rather than having to dump them on you all at once.

None of this says that you *shouldn't* try to think ahead and be clever. In the next chapter, we'll use a bit more design and up-front thinking, and show how that fits in with TDD. But for now, let's plough on mindlessly and just do what the tests tell us to.

1 "Geepaw" Hill, another one of the TDD OGs, has a series of blog posts advocating for taking "Many More Much Smaller Steps (MMMSS)". In this chapter I'm being unrealistically short-sighted for effect, so don't do that! But Geepaw argues that in the real world, when you slice your work into tiny increments, not only do you get there in the end, but you end up delivering business value *faster*.

Wiring Up Our Form to Send a POST Request

At the end of the last chapter, the tests were telling us we weren't able to save the user's input:

```
  File "...goat-book/functional_tests.py", line 40, in
test_can_start_a_todo_list
[...]
AssertionError: False is not true : New to-do item did not appear in table
```

To get it to the server, for now we'll use a standard HTML POST request. A little boring, but also nice and easy to deliver—we can use all sorts of sexy HTML5 and JavaScript later in the book.

To get our browser to send a POST request, we need to do two things:

1. Give the `<input>` element a `name=` attribute.
2. Wrap it in a `<form>` tag[2] with `method="POST"`.

Testing the Contract Between Frontend and Backend

If you remember in the last chapter, we said we wanted to come back and revisit the smoke test of our home page template content. Let's have a quick look at our unit tests:

lists/tests.py

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        response = self.client.get("/")
        self.assertTemplateUsed(response, "home.html")

    def test_renders_homepage_content(self):
        response = self.client.get("/")
        self.assertContains(response, "To-Do")
```

What's important about our home page content? How can we obey both the "don't test constants" rule and the "test behaviour, not implementation" rule?

The specific spelling of the word "To-Do" is not important. As we've just seen, the most important *behaviour* that our home page is enabling, is the ability to submit a to-do item. The way we're going to deliver that is by adding a `<form>` tag with `method="POST"`, and inside that, making sure our `<input>` has a `name="item_text"`.

2 Did you know that you don't need a button to make a form submit? I can't remember when I learned that, but readers have mentioned that it's unusual so I thought I'd draw your attention to it.

Our FTs are telling us that it's not working at a high level, so what unit tests can we write at the lower level? Let's start with the form:

lists/tests.py (ch05l001)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]

    def test_renders_input_form(self):  ❶
        response = self.client.get("/")
        self.assertContains(response, '<form method="POST">')  ❷
```

❶ We change the name of the test.

❷ And we assert on the `<form>` tag specifically.

That gives us:

```
$ python manage.py test
[...]
AssertionError: False is not true : Couldn't find '<form method="POST">' in the
following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <input id="id_new_item" placeholder="Enter a
to-do item" />\n    <table id="id_list_table">\n    </table>\n
</body>\n</html>\n'
```

Let's adjust our template at *lists/templates/home.html*:

lists/templates/home.html (ch05l002)

```
<h1>Your To-Do list</h1>
<form method="POST">
  <input id="id_new_item" placeholder="Enter a to-do item" />
</form>
```

That gives us passing unit tests:

```
OK
```

And next, let's add a test for the `name=` attribute on the `<input>` tag:

lists/tests.py (ch05l003)

```
    def test_renders_input_form(self):
        response = self.client.get("/")
        self.assertContains(response, '<form method="POST">')
        self.assertContains(response, '<input name="item_text"')
```

That gives us this expected failure:

```
[...]
AssertionError: False is not true : Couldn't find '<input name="item_text"' in
the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST">\n      <input
id="id_new_item" placeholder="Enter a to-do item" />\n    </form>\n    <table
id="id_list_table">\n    </table>\n  </body>\n</html>\n'
```

And we fix it like this:

lists/templates/home.html (ch05l004)

```
<h1>Your To-Do list</h1>
<form method="POST">
  <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
</form>
<table id="id_list_table">
```

That gives us passing unit tests:

```
OK
```

The lesson here is that we've tried to identify the "contract" between the frontend and the backend of our site. For our HTML form to work, it needs the form with the right `method`, and the input with the right `name`. Everything else is cosmetic. So that's what we test for in our unit tests.

Debugging Functional Tests

Time to go back to our FT. It gives us a slightly cryptic, unexpected error:

```
$ python functional_tests.py
[...]
Traceback (most recent call last):
  File "...goat-book/functional_tests.py", line 38, in
test_can_start_a_todo_list
    table = self.browser.find_element(By.ID, "id_list_table")
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
```

Oh dear, we're now failing two lines *earlier*, after we submit the form, but before we are able to do the assert. Selenium seems to be unable to find our list table. Why on earth would that happen? Let's take another look at our code:

functional_tests.py

```
        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)

        table = self.browser.find_element(By.ID, "id_list_table")  ❶
        rows = table.find_elements(By.TAG_NAME, "tr")
        self.assertTrue(
            any(row.text == "1: Buy peacock feathers" for row in rows),
            "New to-do item did not appear in table",
        )
```

❶ Our test unexpectedly fails on this line. How do we figure out what's going on?

When a functional test fails with an unexpected failure, there are several things we can do to debug it:

- Add `print` statements to show, for example, what the current page text is.
- Improve the *error message* to show more info about the current state.
- Manually visit the site yourself.
- Use `time.sleep` to pause the test during execution so you can inspect what was happening.[3]

We'll look at all of these over the course of this book, but the `time.sleep` option is the one that leaps to mind with this kind of error in an FT. Let's try it now.

Debugging with time.sleep

Conveniently, we've already got a sleep just before the error occurs; let's just extend it a little:

functional_tests.py (ch05l005)

```
        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)
        time.sleep(10)

        table = self.browser.find_element(By.ID, "id_list_table")
```

3 Another common technique for debugging tests is to use `breakpoint()` to drop into a debugger like `pdb`. This is more useful for *unit* tests rather than FTs though, because in an FT you usually can't step into actual application code. Personally, I only find debuggers useful for really fiddly algorithms, which we won't see in this book.

Depending on how fast Selenium runs on your PC, you may have caught a glimpse of this already, but when we run the FTs again, we've got time to see what's going on: you should see a page that looks like Figure 5-1, with lots of Django debug information.

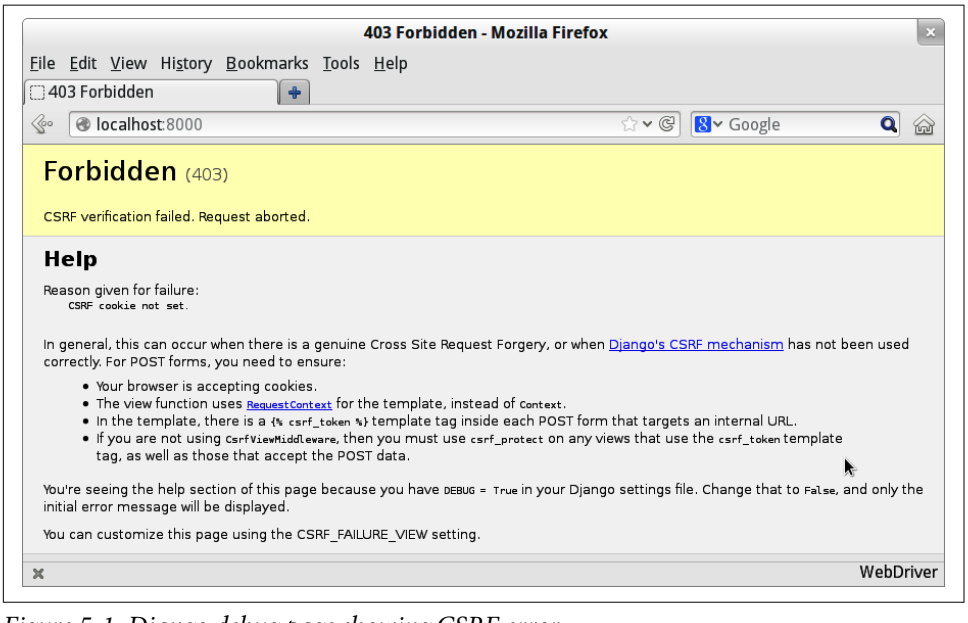


Figure 5-1. Django debug page showing CSRF error

Security: Surprisingly Fun!

If you've never heard of a *cross-site request forgery* (CSRF) exploit, why not look it up now? Like all security exploits, it's entertaining to read about, being an ingenious use of a system in unexpected ways.

When I went to university to get my computer science degree, I signed up for the "security" module out of a sense of duty: *Oh well, it'll probably be very dry and boring, but I suppose I'd better take it. Eat your vegetables, and so forth*. It turned out to be one of the most fascinating modules of the whole course! Absolutely full of the joy of hacking, of the particular mindset it takes to think about how systems can be used in unintended ways.

I want to recommend the textbook from that course, Ross Anderson's *Security Engineering*. It's quite light on pure crypto, but it's absolutely full of interesting discussions of unexpected topics like lock picking, forging bank notes, inkjet printer cartridge economics, and spoofing South African Air Force jets with replay attacks. It's a huge tome, about three inches thick, and I promise you it's an absolute page-turner.

Django's CSRF protection involves placing a little autogenerated unique token into each generated form, to be able to verify that POST requests have definitely come from the form generated by the server. So far, our template has been pure HTML, and in this step we make the first use of Django's template magic. To add the CSRF token, we use a *template tag*, which has the curly-bracket/percent syntax, `{% ... %}`—famous for being the world's most annoying two-key touch-typing combination:

lists/templates/home.html (ch05l006)

```
<form method="POST">
  <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
  {% csrf_token %}
</form>
```

Django will substitute the template tag during rendering with an `<input type="hidden">` containing the CSRF token. Rerunning the functional test will now bring us back to our previous (expected) failure:

```
  File "...goat-book/functional_tests.py", line 40, in
test_can_start_a_todo_list
[...]
AssertionError: False is not true : New to-do item did not appear in table
```

Because our long `time.sleep` is still there, the test will pause on the final screen, showing us that the new item text disappears after the form is submitted, and the page refreshes to show an empty form again. That's because we haven't wired up our server to deal with the POST request yet—it just ignores it and displays the normal home page.

We can put our normal short `time.sleep` back now though:

functional_tests.py (ch05l007)

```
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)

        table = self.browser.find_element(By.ID, "id_list_table")
```

Processing a POST Request on the Server

Because we haven't specified an `action=` attribute in the form, it is submitting back to the same URL it was rendered from by default (i.e., `/`), which is dealt with by our `home_page` function. That's fine for now; let's adapt the view to be able to deal with a POST request.

That means a new unit test for the `home_page` view. Open up *lists/tests.py*, and add a new method to `HomePageTest`:

lists/tests.py (ch05l008)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]
    def test_renders_input_form(self):
        response = self.client.get("/")
        self.assertContains(response, '<form method="POST">')
        self.assertContains(response, '<input name="item_text"')  ❷

    def test_can_save_a_POST_request(self):
        response = self.client.post("/", data={"item_text": "A new list item"})  ❶❷
        self.assertContains(response, "A new list item")  ❸
```

❶ To do a POST, we call `self.client.post` and, as you can see, it takes a `data` argument that contains the form data we want to send.

❷ Notice the echo of the `item_text` name from earlier.[4]

❸ Then we check that the text from our POST request ends up in the rendered HTML.

That gives us our expected fail:

```
$ python manage.py test
[...]
AssertionError: False is not true : Couldn't find 'A new list item' in the
following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST">\n      <input
name="item_text" id="id_new_item" placeholder="Enter a to-do item" />\n
<input type="hidden" name="csrfmiddlewaretoken"
value="[...]
</form>\n    <table id="id_list_table">\n    </table>\n  </body>\n</html>\n'
```

In (slightly exaggerated) TDD style, we can single-mindedly do "the simplest thing that could possibly work" to address this test failure, which is to add an `if` and a new code path for POST requests, with a deliberately silly return value:

lists/views.py (ch05l009)

```
from django.http import HttpResponse
from django.shortcuts import render


def home_page(request):
    if request.method == "POST":  ❶
        return HttpResponse("You submitted: " + request.POST["item_text"])  ❷
    return render(request, "home.html")
```

4 You could even define a constant for this, to make the link more explicit.

❶ `request.method` lets us check whether we got a POST or a GET request.

❷ `request.POST` is a dictionary-like object containing the form data (in this case, the `item_text` value we expect from the form `input` tag).

Fine, that gets our unit tests passing:

```
OK
```

…but it's not really what we want.[5]

And even if we were genuinely hoping this was the right solution, our FTs are here to remind us that this isn't how things are supposed to work:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
```

The list table disappears after the form submission. If you didn't see it in the FT run, try it manually with `runserver`; you'll see something like Figure 5-2.

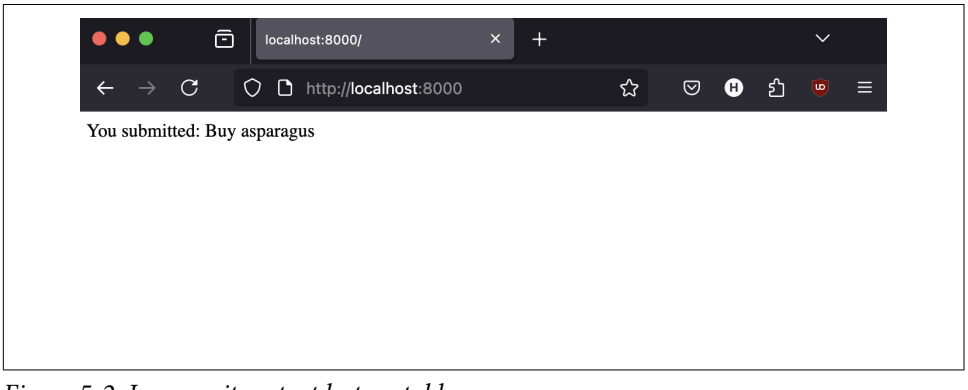


Figure 5-2. I see my item text but no table…

What we really want to do is add the POST submission to the to-do items table in the home page template. We need some sort of way to pass data from our view, to be shown in the template.

5 But we *did* learn about `request.method` and `request.POST`, right? I know it might seem that I'm overdoing it, but doing things in tiny little steps really does have a lot of advantages, and one of them is that you can really think about (or in this case, learn) one thing at a time.

Passing Python Variables to Be Rendered in the Template

We've already had a hint of it, and now it's time to start to get to know the real power of the Django template syntax, which is to pass variables from our Python view code into HTML templates.

Let's start by seeing how the template syntax lets us include a Python object in our template. The notation is `{{ ... }}`, which displays the object as a string:

lists/templates/home.html (ch05l010)

```
<body>
  <h1>Your To-Do list</h1>
  <form method="POST">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
    {% csrf_token %}
  </form>
  <table id="id_list_table">
    <tr><td>{{ new_item_text }}</td></tr>  ❶
  </table>
</body>
```

❶ Here's our template variable. `new_item_text` will be the variable name for the user input we display in the template.

Let's adjust our unit test so that it checks whether we are still using the template:

lists/tests.py (ch05l011)

```
    def test_can_save_a_POST_request(self):
        response = self.client.post("/", data={"item_text": "A new list item"})
        self.assertContains(response, "A new list item")
        self.assertTemplateUsed(response, "home.html")
```

And that will fail as expected:

```
AssertionError: No templates used to render the response
```

Good; our deliberately silly return value is now no longer fooling our tests, so we are allowed to rewrite our view, and tell it to pass the POST parameter to the template. The `render` function takes, as its third argument, a dictionary, which maps template variable names to their values.

In theory, we can use it for the POST case as well as the default GET case, so let's remove the `if request.method == "POST"` and simplify our view right down to:

lists/views.py (ch05l012)

```
def home_page(request):
    return render(
        request,
        "home.html",
        {"new_item_text": request.POST["item_text"]},
    )
```

What do the tests think?

```
ERROR: test_uses_home_template
(lists.tests.HomePageTest.test_uses_home_template)

[...]
    {"new_item_text": request.POST["item_text"]},
                      ~~~~~~~~~~~~^^^^^^^^^^^^^
[...]
django.utils.datastructures.MultiValueDictKeyError: 'item_text'
```

An Unexpected Failure

Oops, an *unexpected failure*.

If you remember the rules for reading tracebacks, you'll spot that it's actually a failure in a *different* test. We got the actual test we were working on to pass, but the unit tests have picked up an unexpected consequence, a regression: we broke the code path where there is no POST request.

This is the whole point of having tests. Yes, perhaps we could have predicted this would happen, but imagine if we'd been having a bad day or weren't paying attention: our tests have just saved us from accidentally breaking our application and, because we're using TDD, we found out immediately. We didn't have to wait for a QA team, or switch to a web browser and click through our site manually, so we can get on with fixing it straight away. Here's how:

lists/views.py (ch05l013)

```
def home_page(request):
    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

We use `dict.get` to supply a default value, for the case where we are doing a normal GET request, when the POST dictionary is empty.

The unit tests should now pass. Let's see what the FTs say:

```
AssertionError: False is not true : New to-do item did not appear in table
```

If your functional tests show you a different error at this point, or at any point in this chapter, complaining about a `StaleElementReferenceException`, you may need to increase the `time.sleep` explicit wait—try two or three seconds instead of one; then read on to the next chapter for a more robust solution.

Improving Error Messages in Tests

Hmm, not a wonderfully helpful error. Let's use another of our FT debugging techniques: improving the error message. This is probably the most constructive technique, because those improved error messages stay around to help debug any future errors:

functional_tests.py (ch05l014)

```
self.assertTrue(
    any(row.text == "1: Buy peacock feathers" for row in rows),
    f"New to-do item did not appear in table. Contents were:\n{table.text}",
)
```

That gives us a more helpful message:

```
AssertionError: False is not true : New to-do item did not appear in table.
Contents were:
Buy peacock feathers
```

Actually, you know what would be even better? Making that assertion a bit less clever! As you may remember from Chapter 4, I was very pleased with myself for using the `any()` function, but one of my early release readers (thanks, Jason!) suggested a much simpler implementation. We can replace all four lines of the `assertTrue` with a single `assertIn`:

functional_tests.py (ch05l015)

```
        self.assertIn("1: Buy peacock feathers", [row.text for row in rows])
```

Much better. You should always be very worried whenever you think you're being clever, because what you're probably being is *overcomplicated*.

Now we get the error message for free:

```
        self.assertIn("1: Buy peacock feathers", [row.text for row in rows])
AssertionError: '1: Buy peacock feathers' not found in ['Buy peacock feathers']
```

Consider me suitably chastened.

If, instead, your FT seems to be saying the table is empty ("not found in ["]"), check your `<input>` tag—does it have the correct `name="item_text"` attribute? And does it have `method="POST"`? Without them, the user's input won't be in the right place in `request.POST`.

The point is that the FT wants us to enumerate list items with a "1:" at the beginning of the first list item.

The fastest way to get that to pass is with another quick "cheating" change to the template:

lists/templates/home.html (ch05l016)

```
<tr><td>1: {{ new_item_text }}</td></tr>
```

When Should You Stop Cheating? DRY Versus Triangulation

People often ask about when it's OK to "stop cheating", and change from an implementation we know to be wrong, to one we're happy with.

One justification is *eliminate duplication*—aka DRY (don't repeat yourself)—which (with some caveats) is a good guideline for any kind of code.

If your test uses a magic constant (like the "1:" in front of our list item), and your application code also uses it, some people say *that* counts as duplication, so it justifies refactoring. Removing the magic constant from the application code usually means you have to stop cheating.

It's a judgement call, but I feel that this is stretching the definition of "repetition" a little, so I often like to use a second technique, which is called *triangulation*: if your tests let you get away with writing "cheating" code that you're not happy with (like returning a magic constant), then *write another test* that forces you to write some better code. That's what we're doing when we extend the FT to check that we get a "2:" when inputting a second list item.

See also "Three Strikes and Refactor" on page 79 for a further note of caution on applying DRY too quickly.

Now we get to the `self.fail('Finish the test!')`. If we get rid of that and finish writing our FT, to add the check for adding a second item to the table (copy and paste is our friend), we begin to see that our first cut solution really isn't going to, um, cut it:

functional_tests.py (ch05l017)

```
# There is still a text box inviting her to add another item.
# She enters "Use peacock feathers to make a fly"
# (Edith is very methodical)
inputbox = self.browser.find_element(By.ID, "id_new_item")
inputbox.send_keys("Use peacock feathers to make a fly")
inputbox.send_keys(Keys.ENTER)
time.sleep(1)

# The page updates again, and now shows both items on her list
table = self.browser.find_element(By.ID, "id_list_table")
rows = table.find_elements(By.TAG_NAME, "tr")
self.assertIn(
    "2: Use peacock feathers to make a fly",
    [row.text for row in rows],
)
self.assertIn(
    "1: Buy peacock feathers",
    [row.text for row in rows],
)

# Satisfied, she goes back to sleep
```

Sure enough, the FTs return an error:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']
```

Three Strikes and Refactor

But before we go further—we've got a bad *code smell*[6] in this FT. We have three almost identical code blocks checking for new items in the list table. When we want to apply the DRY principle, I like to follow the motto *three strikes and refactor*. You can copy and paste code once, and it may be premature to try to remove the duplication it causes, but once you get three occurrences, it's time to tidy up.

Let's start by committing what we have so far. Even though we know our site has a major flaw—it can only handle one list item—it's still further ahead than it was. We may have to rewrite it all, and we may not, but the rule is that before you do any refactoring, always do a commit:

```
$ git diff
# should show changes to functional_tests.py, home.html,
# tests.py and views.py
$ git commit -a
```

Always do a commit before embarking on a refactor.

Onto our functional test refactor. Let's use a helper method—remember, only methods that begin with `test_` will be run as tests, so you can use other methods for your own purposes:

functional_tests.py (ch05l018)

```
    def tearDown(self):
        self.browser.quit()

    def check_for_row_in_list_table(self, row_text):
        table = self.browser.find_element(By.ID, "id_list_table")
        rows = table.find_elements(By.TAG_NAME, "tr")
        self.assertIn(row_text, [row.text for row in rows])

    def test_can_start_a_todo_list(self):
        [...]
```

6 If you've not come across the concept, a "code smell" is something about a piece of code that makes you want to rewrite it. Jeff Atwood has a compilation on his blog, *Coding Horror*. The more experience you gain as a programmer, the more fine-tuned your nose becomes to code smells…

I like to put helper methods near the top of the class, between the `tearDown` and the first test. Let's use it in the FT:

functional_tests.py (ch05l019)

```
        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)
        self.check_for_row_in_list_table("1: Buy peacock feathers")

        # There is still a text box inviting her to add another item.
        # She enters "Use peacock feathers to make a fly"
        # (Edith is very methodical)
        inputbox = self.browser.find_element(By.ID, "id_new_item")
        inputbox.send_keys("Use peacock feathers to make a fly")
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)

        # The page updates again, and now shows both items on her list
        self.check_for_row_in_list_table("2: Use peacock feathers to make a fly")
        self.check_for_row_in_list_table("1: Buy peacock feathers")

        # Satisfied, she goes back to sleep
```

We run the FT again to check that it still behaves in the same way:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']
```

Good. Now we can commit the FT refactor as its own small, atomic change:

```
$ git diff # check the changes to functional_tests.py
$ git commit -a
```

There are a couple more bits of duplication in the FTs, like the repetition of finding the `inputbox`, but they're not as egregious yet, so we'll deal with them later.

Instead, back to work. If we're ever going to handle more than one list item, we're going to need some kind of persistence, and databases are a stalwart solution in this area.

The Django ORM and Our First Model

An object-relational mapper (ORM) is a layer of abstraction for data stored in a database with tables, rows, and columns. It lets us work with databases using familiar object-oriented metaphors that work well with code. Classes map to database tables, attributes map to columns, and an individual instance of the class represents a row of data in the database.

Django comes with an excellent ORM, and writing a unit test that uses it is actually an excellent way of learning it, because it exercises code by specifying how we want it to work.

Let's create a new class in *lists/tests.py*:

lists/tests.py (ch05l020)

```
from django.test import TestCase
from lists.models import Item


class HomePageTest(TestCase):
    [...]


class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
        first_item = Item()
        first_item.text = "The first (ever) list item"
        first_item.save()

        second_item = Item()
        second_item.text = "Item the second"
        second_item.save()

        saved_items = Item.objects.all()
        self.assertEqual(saved_items.count(), 2)

        first_saved_item = saved_items[0]
        second_saved_item = saved_items[1]
        self.assertEqual(first_saved_item.text, "The first (ever) list item")
        self.assertEqual(second_saved_item.text, "Item the second")
```

You can see that creating a new record in the database is a relatively simple matter of creating an object, assigning some attributes, and calling a `.save()` function. Django also gives us an API for querying the database via a class attribute, `.objects`, and we use the simplest possible query, `.all()`, which retrieves all the records for that table. The results are returned as a list-like object called a `QuerySet`, from which we can extract individual objects, and also call further functions, like `.count()`. We then

check the objects as saved to the database, to check whether the right information was saved.

Django's ORM has many other helpful and intuitive features; this might be a good time to skim through the Django tutorial, which has an excellent intro to them.

I've written this unit test in a very verbose style, as a way of introducing the Django ORM. I wouldn't recommend writing your model tests like this "in real life", because it's testing the framework, rather than testing our own code. We'll actually rewrite this test to be much more concise in Chapter 16 (specifically, at "Rewriting the Old Model Test" on page 367).

Unit Tests Versus Integration Tests, and the Database

Some people will tell you that a "real" unit test should never touch the database, and that the test I've just written should be more properly called an "integration" test, because it doesn't *only* test our code, but also relies on an external system—that is, a database.

It's OK to ignore this distinction for now—we have two types of test: the high-level FTs, which test the application from the user's point of view, and these lower-level tests, which test it from the programmer's point of view.

We'll come back to this topic and talk about the differences between unit tests, integration tests, and more in Chapter 27, at the end of the book.

Let's try running the unit test. Here comes another unit-test/code cycle:

```
ImportError: cannot import name 'Item' from 'lists.models'
```

Very well, let's give it something to import from *lists/models.py*. We're feeling confident so we'll skip the `Item = None` step, and go straight to creating a class:

lists/models.py (ch05l021)

```
from django.db import models

# Create your models here.
class Item:
    pass
```

That gets our test as far as:

```
[...]
  File "...goat-book/lists/tests.py", line 25, in
test_saving_and_retrieving_items
    first_item.save()
    ^^^^^^^^^^^^^^^^
AttributeError: 'Item' object has no attribute 'save'
```

To give our `Item` class a `save` method, and to make it into a real Django model, we make it inherit from the `Model` class:

lists/models.py (ch05l022)

```
from django.db import models


class Item(models.Model):
    pass
```

Our First Database Migration

The next thing that happens is a huuuuge traceback, the long and short of which is that there's a problem with the database:

```
django.db.utils.OperationalError: no such table: lists_item
```

In Django, the ORM's job is to model and read and write from database tables, but there's a second system that's in charge of actually *creating* the tables in the database called "migrations". Its job is to let you add, remove, and modify tables and columns, based on changes you make to your *models.py* files.

One way to think of it is as a version control system (VCS) for your database. As we'll see later, it proves particularly useful when we need to upgrade a database that's deployed on a live server.

For now all we need to know is how to build our first database migration, which we do using the `makemigrations` command:[7]

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0001_initial.py
    + Create model Item
$ ls lists/migrations
0001_initial.py  __init__.py  __pycache__
```

7 If you've done a bit of Django before, you may be wondering about when we're going to run "migrate" as well as "makemigrations"? Read on; that's coming up later in the chapter.

If you're curious, you can go and take a look in the migrations file, and you'll see it's a representation of our additions to *models.py*.

In the meantime, we should find that our tests get a little further.

The Test Gets Surprisingly Far

The test actually gets surprisingly far:

```
$ python manage.py test
[...]
    self.assertEqual(first_saved_item.text, "The first (ever) list item")
                     ^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'Item' object has no attribute 'text'
```

That's a full eight lines later than the last failure—we've been all the way through saving the two `Items`, and we've checked that they're saved in the database, but Django just doesn't seem to have "remembered" the `.text` attribute.

If you're new to Python, you might have been surprised that we were allowed to assign the `.text` attribute at all. In a language like Java, you would probably get a compilation error. Python is more relaxed.

Classes that inherit from `models.Model` will map to tables in the database. By default, they get an autogenerated `id` attribute, which will be a primary key column[8] in the database, but you have to define any other columns and attributes you want explicitly. Here's how we set up a text column:

lists/models.py (ch05l024)

```
class Item(models.Model):
    text = models.TextField()
```

Django has many other field types, like `IntegerField`, `CharField`, `DateField`, and so on. I've chosen `TextField` rather than `CharField` because the latter requires a length restriction, which seems arbitrary at this point. You can read more on field types in the Django tutorial and in the documentation.

8 Database tables usually have a special column called a "primary key", which is the unique identifier for each row in the table. It's worth brushing up on a *tiny* bit of relational database theory, if you're not familiar with the concept or why it's useful. The top three articles I found when searching for "introduction to databases" all seemed pretty good, at the time of writing.

A New Field Means a New Migration

Running the tests gives us another database error:

```
django.db.utils.OperationalError: table lists_item has no column named text
```

It's because we've added another new field to our database, which means we need to create another migration. Nice of our tests to let us know!

Let's try it:

```
$ python manage.py makemigrations
It is impossible to add a non-nullable field 'text' to item without specifying
a default. This is because the database needs something to populate existing
rows.
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null
value for this column)
 2) Quit and manually define a default value in models.py.
Select an option:2
```

Ah. It won't let us add the column without a default value. Let's pick option 2 and set a default in *models.py*. I think you'll find the syntax reasonably self-explanatory:

lists/models.py (ch05l025)

```
class Item(models.Model):
    text = models.TextField(default="")
```

And now the migration should complete:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0002_item_text.py
    + Add field text to item
```

So, two new lines in *models.py*, two database migrations, and as a result, the `.text` attribute on our model objects is now recognised as a special attribute, so it does get saved to the database, and the tests pass:

```
$ python manage.py test
[...]

Ran 4 tests in 0.010s
OK
```

So let's do a commit for our first ever model!

```
$ git status # see tests.py, models.py, and 2 untracked migrations
$ git diff # review changes to tests.py and models.py
$ git add lists
$ git commit -m "Model for list Items and associated migration"
```

Saving the POST to the Database

So, we have a model; now we need to use it!

Let's adjust the test for our home page POST request, and say we want the view to save a new item to the database instead of just passing it through to its response. We can do that by adding three new lines to the existing test called `test_can_save_a_POST_request`:

lists/tests.py (ch05l027)

```
    def test_can_save_a_POST_request(self):
        response = self.client.post("/", data={"item_text": "A new list item"})

        self.assertEqual(Item.objects.count(), 1)  ❶
        new_item = Item.objects.first()  ❷
        self.assertEqual(new_item.text, "A new list item")  ❸

        self.assertContains(response, "A new list item")
        self.assertTemplateUsed(response, "home.html")
```

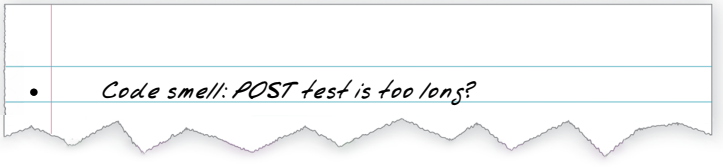
❶ We check that one new `Item` has been saved to the database. `objects.count()` is a shorthand for `objects.all().count()`.

❷ `objects.first()` is the same as doing `objects.all()[0]`, except it will return `None` if there are no objects.[9]

❸ We check that the item's text is correct.

9 You can also use `objects.get()`, which will immediately raise an exception if there are no objects in the database, or if there are more than one. On the plus side you get a more immediate failure, and you get warned if there are too many objects. The downside is that I find it slightly less readable. As so often, it's a trade-off.

This test is getting a little long-winded. It seems to be testing lots of different things. That's another *code smell*—a long unit test either needs to be broken into two, or it may be an indication that the thing you're testing is too complicated. Let's add that to a little to-do list of our own, perhaps on a piece of scrap paper:



An Alternative Testing Strategy: Staying at the HTTP Level

It's a very common pattern in Django to test POST views by asserting on the side effects, as seen in the database. Sandi Metz, a TDD legend from the Ruby world, puts it like this: "test commands via public side effects".[10]

But is the database really a public API? That's arguable. Certainly it's at a different level of abstraction, or a different conceptual "layer" in the application, to the HTTP requests we're working with in our current unit tests.

If you wanted to write our tests in a way that stays at the HTTP level—that treats the application as more of an "opaque box"—you can prove to yourself that to-do items are persisted, by sending more than one:

lists/tests/tests.py

```
    def test_can_save_multiple_items(self):
        self.client.post("/", data={"item_text": "first item"})
        response = self.client.post("/", data={"item_text": "second item"})
        self.assertContains(response, "first item")
        self.assertContains(response, "second item")
```

If you feel like going off road, why not give it a try?

10 This advice is in her talk "The Magic Tricks of Testing", which I highly recommend watching.

Writing things down on a scratchpad like this reassures us that we won't forget them, so we are comfortable getting back to what we were working on. We rerun the tests and see an expected failure:

```
        self.assertEqual(Item.objects.count(), 1)
    AssertionError: 0 != 1
```

Let's adjust our view:

lists/views.py (ch05l028)

```
from django.shortcuts import render
from lists.models import Item


def home_page(request):
    item = Item()
    item.text = request.POST.get("item_text", "")
    item.save()

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

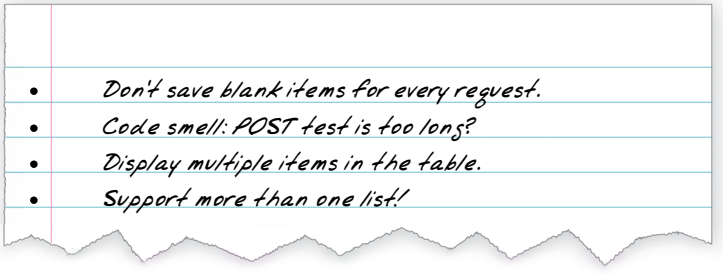
I've coded a very naive solution and you can probably spot a very obvious problem, which is that we're going to be saving empty items with every request to the home page. Let's add that to our list of things to fix later. You know, along with the painfully obvious fact that we currently have no way at all of having different lists for different people. That we'll keep ignoring for now.

Remember, I'm not saying you should always ignore glaring problems like this in "real life". Whenever we spot problems in advance, there's a judgement call to make over whether to stop what you're doing and start again, or leave them until later. Sometimes finishing off what you're doing is still worth it, and sometimes the problem may be so major as to warrant a stop and rethink.

Let's see how the unit tests get on…

```
Ran 4 tests in 0.010s

OK
```

They pass! Good. Let's have a little look at our scratchpad. I've added a couple of the other things that are on our mind:



Let's start with the first scratchpad item: "Don't save blank items for every request". We could tack on an assertion to an existing test, but it's best to keep unit tests to testing one thing at a time, so let's add a new one:

lists/tests.py (ch05l029)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]

    def test_can_save_a_POST_request(self):
        [...]

    def test_only_saves_items_when_necessary(self):
        self.client.get("/")
        self.assertEqual(Item.objects.count(), 0)
```

That gives us a `1 != 0` failure. Let's fix it by bringing the `if request.method` check back and putting the `Item` creation in there:

lists/views.py (ch05l030)

```
def home_page(request):
    if request.method == "POST":  ❶
        item = Item()
        item.text = request.POST["item_text"]  ❷
        item.save()

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

❶ We bring back the `request.method` check.

❷ And we can switch from using `request.POST.get()` to `request.POST[]` with square brackets, because we know for sure that the `item_text` key should be in there, and it's better to fail hard if it isn't.

And that gets the test passing:

```
Ran 5 tests in 0.010s

OK
```

Redirect After a POST

But, yuck—those duplicated `request.POST` accesses are making me pretty unhappy. Thankfully we are about to have the opportunity to fix it. A view function has two jobs: processing user input and returning an appropriate response. We've taken care of the first part, which is saving the user's input to the database, so now let's work on the second part.

Always redirect after a POST, they say, so let's do that. Once again we change our unit test for saving a POST request: instead of expecting a response with the item in it, we want it to expect a redirect back to the home page.

lists/tests.py (ch05l031)

```
    def test_can_save_a_POST_request(self):
        response = self.client.post("/", data={"item_text": "A new list item"})

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, "A new list item")

        self.assertRedirects(response, "/")  ❶

    def test_only_saves_items_when_necessary(self):
        [...]
```

❶ We no longer expect a response with HTML content rendered by a template, so we lose the `assertContains` calls that looked at that. Instead, we use Django's `assertRedirects` helper, which checks that we return an HTTP 302 redirect, back to the home URL.

That gives us this expected failure:

```
AssertionError: 200 != 302 : Response didn't redirect as expected: Response
code was 200 (expected 302)
```

We can now tidy up our view substantially:

lists/views.py (ch05l032)

```
from django.shortcuts import redirect, render
from lists.models import Item


def home_page(request):
    if request.method == "POST":
        item = Item()
        item.text = request.POST["item_text"]
        item.save()
        return redirect("/")

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

And the tests should now pass:

```
Ran 5 tests in 0.010s

OK
```

We're at green; time for a little refactor!

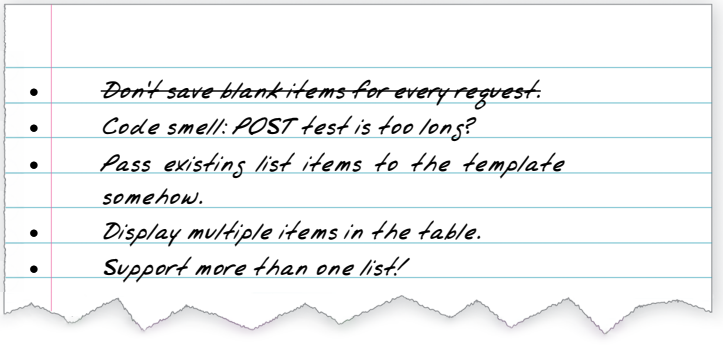
Let's have a look at *views.py* and see what opportunities for improvement there might be:

lists/views.py

```
def home_page(request):
    if request.method == "POST":
        item = Item()  ❶
        item.text = request.POST["item_text"]  ❶
        item.save()  ❶
        return redirect("/")

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},  ❷
    )
```

❶ There's a quicker way to do these three lines with `.objects.create()`.

❷ This line doesn't seem quite right now; in fact, it won't work at all. Let's make a note on our scratchpad to sort out passing list items to the template. It's actually closely related to "Display multiple items", so we'll put it just before that one:



And here's the refactored version of *views.py* using the `.objects.create()` helper method that Django provides, for one-line creation of objects:

lists/views.py (ch05l033)

```
def home_page(request):
    if request.method == "POST":
        Item.objects.create(text=request.POST["item_text"])
        return redirect("/")

    return render(
        request,
        "home.html",
        {"new_item_text": request.POST.get("item_text", "")},
    )
```

Better Unit Testing Practice: Each Test Should Test One Thing

Let's address the "POST test is too long" code smell.

Good unit testing practice says that each test should only test one thing. The reason is that it makes it easier to track down bugs. Having multiple assertions in a test means that, if the test fails on an early assertion, you don't know what the statuses of the later assertions are. As we'll see in the next chapter, if we ever break this view accidentally, we want to know whether it's the saving of objects that's broken, or the type of response.

You may not always write perfect unit tests with single assertions on your first go, but now feels like a good time to separate out our concerns:

lists/tests.py (ch05l034)

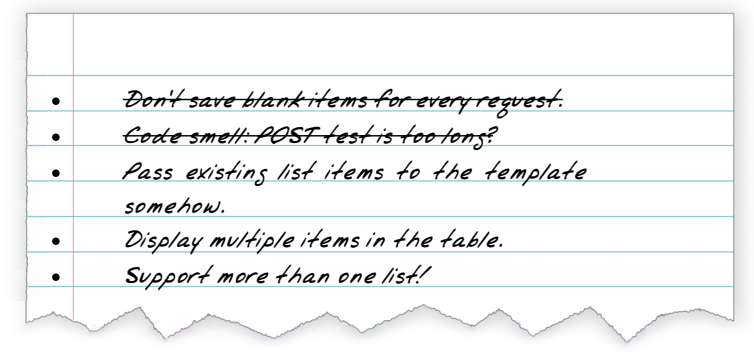
```
    def test_can_save_a_POST_request(self):
        self.client.post("/", data={"item_text": "A new list item"})
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.first()
        self.assertEqual(new_item.text, "A new list item")

    def test_redirects_after_POST(self):
        response = self.client.post("/", data={"item_text": "A new list item"})
        self.assertRedirects(response, "/")
```

And we should now see six tests pass instead of five:

```
Ran 6 tests in 0.010s

OK
```

Rendering Items in the Template

Much better! Back to our to-do list:



Crossing things off the list is almost as satisfying as seeing tests pass!

The third and fourth items are the last of the "easy" ones. Our view now does the right thing for POST requests; it saves new list items to the database. Now we want GET requests to load all currently existing list items, and pass them to the template for rendering. Let's have a new unit test for that:

lists/tests.py (ch05l035)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]
    def test_renders_input_form(self):
        [...]

    def test_displays_all_list_items(self):
        Item.objects.create(text="itemey 1")
        Item.objects.create(text="itemey 2")

        response = self.client.get("/")

        self.assertContains(response, "itemey 1")
        self.assertContains(response, "itemey 2")

    def test_can_save_a_POST_request(self):
        [...]
```

Arrange-Act-Assert or Given-When-Then

Did you notice the use of whitespace in this test? I'm visually separating out the code into three blocks:

lists/tests.py

```
    def test_displays_all_list_items(self):
        Item.objects.create(text="itemey 1")  ❶
        Item.objects.create(text="itemey 2")  ❶

        response = self.client.get("/")  ❷

        self.assertContains(response, "itemey 1")  ❸
        self.assertContains(response, "itemey 2")  ❸
```

❶ Arrange: where we set up the data we need for the test.

❷ Act: where we call the code under test

❸ Assert: where we check on the results

This isn't obligatory, but it's a common convention, and it does help see the structure of the test.

Another popular way to talk about this structure is *given-when-then*:

- *Given* the database contains our list with two items,
- *When* I make a GET request for our list,
- *Then* I see the both items in our list.

This latter phrasing comes from the world of behaviour-driven development (BDD), and I actually prefer it somewhat. You can see that it encourages phrasing things in a more natural way, and we're gently nudged to think of things in terms of behaviour and the perspective of the user.

That fails as expected:

```
AssertionError: False is not true : Couldn't find 'itemey 1' in the following
response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
[...]
```

The Django template syntax has a tag for iterating through lists, `{% for .. in .. %}`; we can use it like this:

lists/templates/home.html (ch05l036)

```
<table id="id_list_table">
  {% for item in items %}
    <tr><td>1: {{ item.text }}</td></tr>
  {% endfor %}
</table>
```

This is one of the major strengths of the templating system. Now the template will render with multiple `<tr>` rows, one for each item in the variable `items`. Pretty neat! I'll introduce a few more bits of Django template magic as we go, but at some point you'll want to go and read up on the rest of them in the Django docs.

Just changing the template doesn't get our tests to green; we need to actually pass the items to it from our home page view:

lists/views.py (ch05l037)

```
def home_page(request):
    if request.method == "POST":
        Item.objects.create(text=request.POST["item_text"])
        return redirect("/")

    items = Item.objects.all()
    return render(request, "home.html", {"items": items})
```

That does get the unit tests to pass. Moment of truth…will the functional test pass?

```
$ python functional_tests.py
[...]
AssertionError: 'To-Do' not found in 'OperationalError at /'
```

Oops, apparently not. Let's use another FT debugging technique, and it's one of the most straightforward: manually visiting the site! Open up *http://localhost:8000* in your web browser, and you'll see a Django debug page saying "no such table: lists_item", as in Figure 5-3.

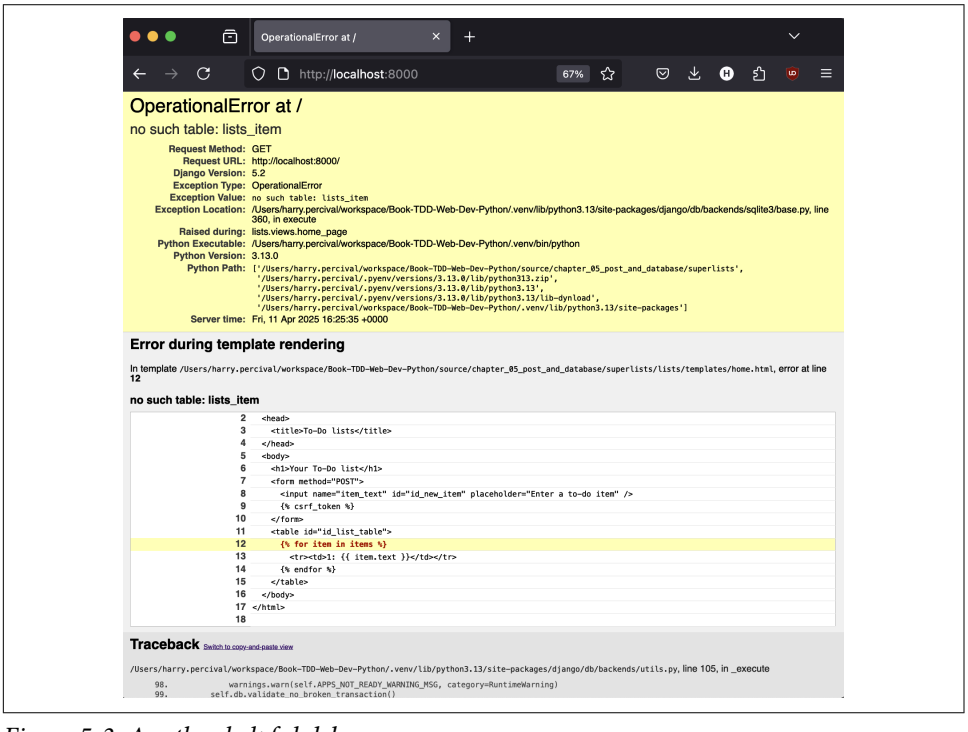


Figure 5-3. Another helpful debug message

Creating Our Production Database with migrate

So, we've got another helpful error message from Django, which is basically complaining that we haven't set up the database properly. How come everything worked fine in the unit tests, I hear you ask? Because Django creates a special *test database* for unit tests; it's one of the magical things that Django's `TestCase` does.

To set up our "real" database, we need to explicitly create it. SQLite databases are just a file on disk, and you'll see in *settings.py* that Django, by default, will just put it in a file called *db.sqlite3* in the base project directory:

superlists/settings.py

```
[...]
# Database
# https://docs.djangoproject.com/en/5.2/ref/settings/#databases

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
    }
}
```

We've told Django everything it needs to create the database, first via *models.py* and then when we created the migrations file. To actually apply it to creating a real database, we use another Django Swiss Army knife *manage.py* command, `migrate`:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, lists, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying lists.0001_initial... OK
  Applying lists.0002_item_text... OK
  Applying sessions.0001_initial... OK
```

It seems to be doing quite a lot of work! That's because it's the first ever migration, and Django is creating tables for all its built-in "batteries included" apps, like the admin site and the built-in auth modules. We don't need to pay attention to them for now. But you can see our `lists.0001_initial` and `lists.0002_item_text` in there!

At this point, you can refresh the page on *localhost* and see that the error is gone. Let's try running the functional tests again:[11]

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers', '1: Use peacock feathers to make a fly']
```

So close! We just need to get our list numbering right. Another awesome Django template tag, `forloop.counter`, will help here:

lists/templates/home.html (ch05l038)

```
{% for item in items %}
  <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
{% endfor %}
```

11 If you get a different error at this point, try restarting your dev server—it may have gotten confused by the changes to the database happening under its feet.

If you try it again, you should now see the FT gets to the end:

```
$ python functional_tests.py
.
 ---------------------------------------------------------------------
Ran 1 test in 5.036s

OK
```

Hooray! But, as it's running, you may notice something is amiss, like in Figure 5-4.

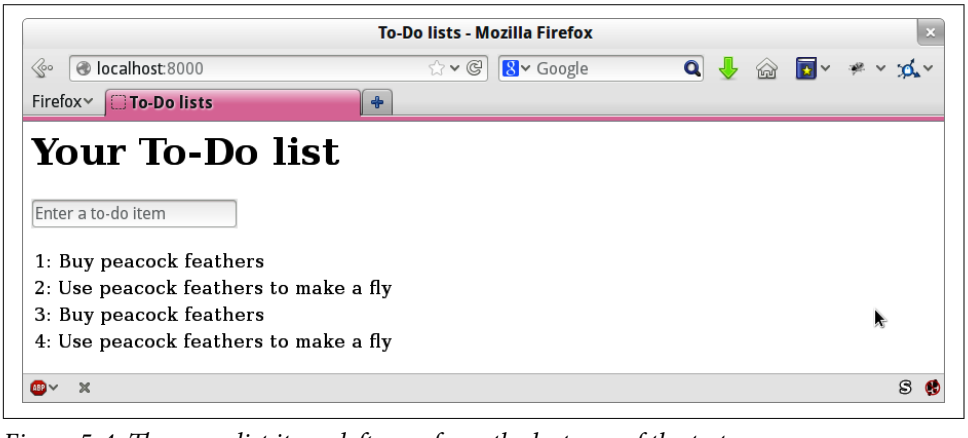


Figure 5-4. There are list items left over from the last run of the test

Oh dear. It looks like previous runs of the test are leaving stuff lying around in our database. In fact, if you run the tests again, you'll see it gets worse:

```
1: Buy peacock feathers
2: Use peacock feathers to make a fly
3: Buy peacock feathers
4: Use peacock feathers to make a fly
5: Buy peacock feathers
6: Use peacock feathers to make a fly
```

Grrr. We're so close! We're going to need some kind of automated way of tidying up after ourselves. For now, if you feel like it, you can do it manually by deleting the database and re-creating it fresh with `migrate` (you'll need to shut down your Django server first):

```
$ rm db.sqlite3
$ python manage.py migrate --noinput
```

And then (after restarting your server!) reassure yourself that the FT still passes.

Apart from that little bug in our functional testing, we've got some code that's more or less working. Let's do a commit.

Start by doing a `git status` and a `git diff`, and you should see changes to *home.html*, *tests.py*, and *views.py*. Let's add them:

```
$ git add lists
$ git commit -m "Redirect after POST, and show all items in template"
```

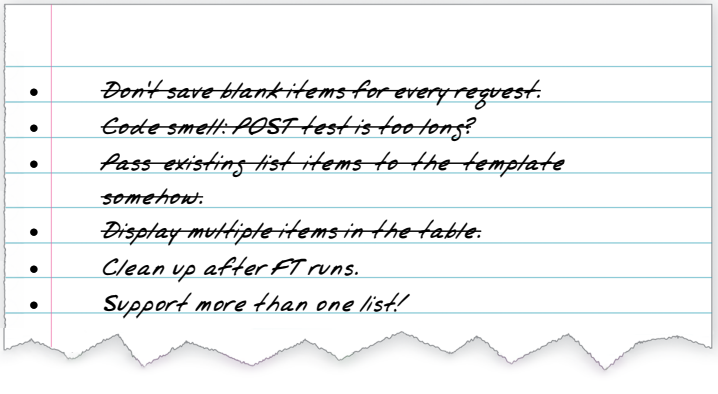
You might find it useful to add markers for the end of each chapter, like **`git tag end-of-chapter-05`**.

Recap

Where are we? How is progress on our app, and what have we learned?

- We've got a form set up to add new items to the list using POST.
- We've set up a simple model in the database to save list items.
- We've learned about creating database migrations, both for the test database (where they're applied automatically) and for the real database (where we have to apply them manually).
- We've used our first couple of Django template tags: `{% csrf_token %}` and the `{% for ... endfor %}` loop.
- And we've used two different FT debugging techniques: `time.sleeps`, and improving the error messages.

But we've got a couple of items on our own to-do list, namely getting the FT to clean up after itself, and perhaps more critically, adding support for more than one list:



I mean, we *could* ship the site as it is, but people might find it strange that the entire human population has to share a single to-do list. I suppose it might get people to stop and think about how connected we all are to one another, how we all share a common destiny here on Spaceship Earth, and how we must all work together to solve the global problems that we face.

But in practical terms, the site wouldn't be very useful.

Ah well.

Useful TDD Concepts

Regression
: When a change unexpectedly breaks some aspect of the application that used to work.

Unexpected failure
: When a test fails in a way we weren't expecting. This either means that we've made a mistake in our tests, or that the tests have helped us find a regression, and we need to fix something in our code.

Triangulation
: Adding a test case with a new specific example for some existing code, to justify generalising the implementation (which may be a "cheat" until that point).

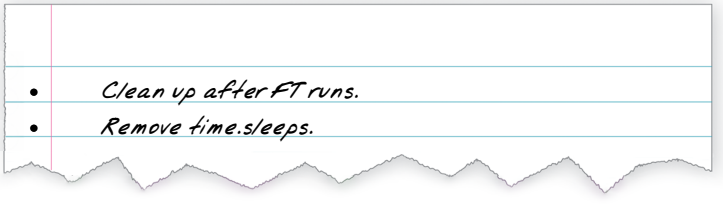
Three strikes and refactor
: A rule of thumb for when to remove duplication from code. When two pieces of code look very similar, it often pays to wait until you see a third use case, so that you're more sure about what part of the code really is the common, reusable part to refactor out.

The scratchpad to-do list
: A place to write down things that occur to us as we're coding, so that we can finish up what we're doing and come back to them later. Love a good old-fashioned piece of paper now and again!

CHAPTER 6

Improving Functional Tests: Ensuring Isolation and Removing Magic Sleeps

Before we dive in and fix our single-global-list problem, let's take care of a couple of housekeeping items. At the end of the last chapter, we made a note that different test runs were interfering with each other, so we'll fix that. I'm also not happy with all these `time.sleep`s peppered through the code; they seem a bit unscientific, so we'll replace them with something more reliable:



Both of these changes will be moving us towards testing "best practices", making our tests more deterministic and more reliable.

Ensuring Test Isolation in Functional Tests

We ended the last chapter with a classic testing problem: how to ensure *isolation* between tests. Each run of our functional tests (FTs) left list items lying around in the database, and that interfered with the test results when next running the tests.

When we run *unit* tests, the Django test runner automatically creates a brand new test database (separate from the real one), which it can safely reset before each individual test is run, and then thrown away at the end. But our FTs currently run against the "real" database, *db.sqlite3*.

One way to tackle this would be to "roll our own" solution, and add some code to *functional_tests.py*, which would do the cleaning up. The `setUp` and `tearDown` methods are perfect for this sort of thing.

But as this is a common problem, Django supplies a test class called `LiveServerTestCase` that addresses this issue. It will automatically create a test database (just like in a unit test run) and start up a development server for the FTs to run against. Although as a tool it has some limitations, which we'll need to work around later, it's dead useful at this stage, so let's check it out.

`LiveServerTestCase` expects to be run by the Django test runner using *manage.py*, which will run tests from any files whose name begins with *test_*. To keep things neat and tidy, let's make a folder for our FTs, so that it looks a bit like an app. All Django needs is for it to be a valid Python package directory (i.e., one with a *___init___.py* in it):

```
$ mkdir functional_tests
$ touch functional_tests/__init__.py
```

Now we want to *move* our functional tests, from being a standalone file called *functional_tests.py*, to being the *tests.py* of the `functional_tests` app. We use `git mv` so that Git keeps track of the fact that this is the same file and should have a single history.

```
$ git mv functional_tests.py functional_tests/tests.py
$ git status # shows the rename to functional_tests/tests.py and __init__.py
```

At this point, your directory tree should look like this:

```
.
├── db.sqlite3
├── functional_tests
│   ├── __init__.py
│   └── tests.py
├── lists
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   ├── 0002_item_text.py
│   │   └── __init__.py
│   ├── models.py
│   ├── templates
│   │   └── home.html
│   ├── tests.py
│   └── views.py
├── manage.py
└── superlists
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

functional_tests.py is gone, and has turned into *functional_tests/tests.py*. Now, whenever we want to run our FTs, instead of running `python functional_tests.py`, we will use `python manage.py test functional_tests`.

You could mix your functional tests into the tests for the `lists` app. I tend to prefer keeping them separate, because FTs usually have cross-cutting concerns that run across different apps. FTs are meant to see things from the point of view of your users, and your users don't care about how you've split work between different apps!

Now, let's edit *functional_tests/tests.py* and change our `NewVisitorTest` class to make it use `LiveServerTestCase`:

functional_tests/tests.py (ch06l001)

```
from django.test import LiveServerTestCase
from selenium import webdriver
[...]


class NewVisitorTest(LiveServerTestCase):
    def setUp(self):
        [...]
```

Next, instead of hardcoding the visit to localhost port `8000`, `LiveServerTestCase` gives us an attribute called `live_server_url`:

functional_tests/tests.py (ch06l002)

```
    def test_can_start_a_todo_list(self):
        # Edith has heard about a cool new online to-do app.
        # She goes to check out its homepage
        self.browser.get(self.live_server_url)
```

We can also remove the `if __name__ == '__main__'` from the end if we want, as we'll be using the Django test runner to launch the FT.

Now we are able to run our functional tests using the Django test runner, by telling it to run just the tests for our new `functional_tests` app:

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
Found 1 test(s).
System check identified no issues (0 silenced).
.
 ----------------------------------------------------------------------
Ran 1 test in 10.519s

OK
Destroying test database for alias 'default'...
```

When I ran this test today, I ran into the Firefox upgrade pop-up. Just a little reminder, in case you happen to see it too, we talked about it in Chapter 1 in a little sidebar.

The FT still passes, reassuring us that our refactor didn't break anything. You'll also notice that if you run the tests a second time, there aren't any old list items lying around from the previous test—it has cleaned up after itself. Success! We should commit it as an atomic change:

```
$ git status # functional_tests.py renamed + modified, new __init__.py
$ git add functional_tests
$ git diff --staged
$ git commit  # msg eg "make functional_tests an app, use LiveServerTestCase"
```

Running Just the Unit Tests

Now if we run `manage.py test`, Django will run both the functional and the unit tests:

```
$ python manage.py test
Creating test database for alias 'default'...
Found 8 test(s).
System check identified no issues (0 silenced).
........
 ----------------------------------------------------------------------
Ran 8 tests in 10.859s

OK
Destroying test database for alias 'default'...
```

To run just the unit tests, we can specify that we want to only run the tests for the `lists` app:

```
$ python manage.py test lists
Creating test database for alias 'default'...
Found 7 test(s).
System check identified no issues (0 silenced).
.......
 ----------------------------------------------------------------------
Ran 7 tests in 0.009s

OK
Destroying test database for alias 'default'...
```

Useful Commands Updated

To run the functional tests

```
python manage.py test functional_tests
```

To run the unit tests

```
python manage.py test lists
```

What to do if I say "run the tests", and you're not sure which ones I mean? Have another look at the flowchart at the end of Chapter 4, and try to figure out where we are. As a rule of thumb, we usually only run the FTs once all the unit tests are passing, so if in doubt, try both!

On Implicit and Explicit Waits, and Magic time.sleeps

Let's talk about the `time.sleep` in our FT:

functional_tests/tests.py

```
        # When she hits enter, the page updates, and now the page lists
        # "1: Buy peacock feathers" as an item in a to-do list table
        inputbox.send_keys(Keys.ENTER)
        time.sleep(1)

        self.check_for_row_in_list_table("1: Buy peacock feathers")
```

This is what's called an "explicit wait". That's in contrast with "implicit waits": in certain cases, Selenium tries to wait "automatically" for you when it thinks the page is loading. It even provides a method called `implicitly_wait` that lets you control how long it will wait if you ask it for an element that doesn't seem to be on the page yet.

In fact, in the first edition of this book, I was able to rely entirely on implicit waits. The problem is that implicit waits are always a little flakey, and with the release of Selenium 4, implicit waits were disabled by default. At the same time, the general opinion from the Selenium team is that implicit waits are just a bad idea, and should be avoided.

So this edition has explicit waits from the very beginning. But the problem is that those `time.sleep`s have their own issues.

Currently we're waiting for one second, but who's to say that's the right amount of time? For most tests we run against our own machine, one second is way too long, and it's going to really slow down our FT runs. 0.1s would be fine. But the problem is that if you set it that low, every so often you're going to get a spurious failure because, for whatever reason, the laptop was being a bit slow just then. And even

at one second, there's still a chance of random failures that don't indicate a real problem—and false positives in tests are a real annoyance.[1]

Unexpected `NoSuchElementException` and `StaleElementException` errors are often a sign that you need an explicit wait.

So let's replace our sleeps with a tool that will wait for just as long as is needed, up to a nice long timeout to catch any glitches. We'll rename `check_for_row_in_list_table` to `wait_for_row_in_list_table`, and add some polling/retry logic to it:

functional_tests/tests.py (ch06l004)

```
[...]
from selenium.common.exceptions import WebDriverException
import time

MAX_WAIT = 5  ❶


class NewVisitorTest(LiveServerTestCase):
    def setUp(self):
        [...]
    def tearDown(self):
        [...]

    def wait_for_row_in_list_table(self, row_text):
        start_time = time.time()
        while True:  ❷
            try:
                table = self.browser.find_element(By.ID, "id_list_table")  ❸
                rows = table.find_elements(By.TAG_NAME, "tr")
                self.assertIn(row_text, [row.text for row in rows])
                return  ❹
            except (AssertionError, WebDriverException):  ❺
                if time.time() - start_time > MAX_WAIT:  ❻
                    raise  ❻
                time.sleep(0.5)  ❺
```

1 There's lots more on this in an article by Martin Fowler.

❶ We'll use a constant called `MAX_WAIT` to set the maximum amount of time we're prepared to wait. Five seconds should be enough to catch any glitches or random slowness.

❷ Here's the loop, which will keep going forever, unless we get to one of two possible exit routes.

❸ Here are our three lines of assertions from the old version of the method.

❹ If we get through them, and our assertion passes, we return from the function and escape the loop.

❺ But if we catch an exception, we wait a short amount of time and loop around to retry. There are two types of exceptions we want to catch: `WebDriverException` for when the page hasn't loaded and Selenium can't find the table element on the page; and `AssertionError` for when the table is there, but it's perhaps a table from before the page reloads, so it doesn't have our row in yet.

❻ Here's our second escape route. If we get to this point, that means our code kept raising exceptions every time we tried it until we exceeded our timeout. So this time, we reraise the exception and let it bubble up to our test, and most likely end up in our traceback, telling us why the test failed.

Are you thinking this code is a little ugly, and makes it a bit harder to see exactly what we're doing? I agree. Later on (Example 13-12), we'll refactor out a general `wait_for` helper, to separate the timing and reraising logic from the test assertions. But we'll wait until we need it in multiple places.

If you've used Selenium before, you may know that it has a few helper functions to conduct waits. I'm not a big fan of them, though not for any objective reason really. Over the course of the book, we'll build a couple of wait helper tools, which I think will make for nice and readable code. But of course you should check out the homegrown Selenium waits in your own time, and see if you prefer them.

Now we can rename our method calls, and remove the magic `time.sleeps`:

functional_tests/tests.py (ch06l005)

```
    [...]
    # When she hits enter, the page updates, and now the page lists
    # "1: Buy peacock feathers" as an item in a to-do list table
    inputbox.send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table("1: Buy peacock feathers")

    # There is still a text box inviting her to add another item.
    # She enters "Use peacock feathers to make a fly"
    # (Edith is very methodical)
    inputbox = self.browser.find_element(By.ID, "id_new_item")
    inputbox.send_keys("Use peacock feathers to make a fly")
    inputbox.send_keys(Keys.ENTER)

    # The page updates again, and now shows both items on her list
    self.wait_for_row_in_list_table("2: Use peacock feathers to make a fly")
    self.wait_for_row_in_list_table("1: Buy peacock feathers")
    [...]
```

And rerun the tests:

```
$ python manage.py test
Creating test database for alias 'default'...
Found 8 test(s).
System check identified no issues (0 silenced).
........
 ----------------------------------------------------------------------
Ran 8 tests in 4.552s

OK
Destroying test database for alias 'default'...
```

Hooray we're back to passing, and notice we've shaved a few of seconds off the execution time too. That might not seem like a lot right now, but it all adds up.

Just to check we've done the right thing, let's deliberately break the test in a couple of ways and see some errors. First, let's try searching for some text that we know isn't there, and check that we get the expected error:

functional_tests/tests.py (ch06l006)

```
def wait_for_row_in_list_table(self, row_text):
    [...]
        rows = table.find_elements(By.TAG_NAME, "tr")
        self.assertIn("foo", [row.text for row in rows])
        return
```

We see we still get a nice self-explanatory test failure message:

```
    self.assertIn("foo", [row.text for row in rows])
AssertionError: 'foo' not found in ['1: Buy peacock feathers']
```

Did you get a bit bored waiting five seconds for the test to fail? That's one of the downsides of explicit waits. There's a tricky trade-off between waiting long enough that little glitches don't throw you, versus waiting so long that expected failures are painfully slow to watch. Making `MAX_WAIT` configurable so that it's fast in local dev, but more conservative on continuous integration (CI) servers can be a good idea. See Chapter 25 for an introduction to CI.

Let's put that back the way it was and break something else:

functional_tests/tests.py (ch06l007)

```
        try:
            table = self.browser.find_element(By.ID, "id_nothing")
            rows = table.find_elements(By.TAG_NAME, "tr")
            self.assertIn(row_text, [row.text for row in rows])
            return
        [...]
```

Sure enough, we get the errors for when the page doesn't contain the element we're looking for too:

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_nothing"]; For documentation on this error, [...]
```

Everything seems to be in order. Let's put our code back to the way it should be, and do one final test run:

```
$ python manage.py test
[...]
OK
```

Great. With that little interlude over, let's crack on with getting our application actually working for multiple lists. Don't forget to commit first!

Testing "Best Practices" Applied in this Chapter

Ensuring test isolation and managing global state
: Different tests shouldn't affect one another. This means we need to reset any permanent state at the end of each test. Django's test runner helps us do this by creating a test database, which it wipes clean in between each test.

Avoid "magic" sleeps
: Whenever we need to wait for something to load, it's always tempting to throw in a quick-and-dirty `time.sleep`. But the problem is that the length of time we wait is always a bit of a shot in the dark, either too short and too vulnerable to spurious failures, or too long and it'll slow down our test runs. Prefer a retry loop that polls our app and moves on as soon as possible.

Don't rely on Selenium's implicit waits
: Selenium does theoretically do some "implicit" waits, but the implementation varies between browsers, and is not always reliable. "Explicit is better than implicit", as the Zen of Python says,[2] so prefer explicit waits.

2 `python -c "import this"`

CHAPTER 7

Working Incrementally

Now let's address our real problem, which is that our design only allows for one global list. In this chapter I'll demonstrate a critical TDD technique: how to adapt existing code using an incremental, step-by-step process that takes you from working state to working state. Testing Goat, not Refactoring Cat!

Small Design When Necessary

Let's have a think about how we want support for multiple lists to work.

At the moment, the only URL for our site is the home page, and that's why there's only one global list. The most obvious way to support multiple lists is to say that each list gets its own URL, so that people can start multiple lists, or so that different people can have different lists. How might that work?

Not Big Design Up Front

TDD is closely associated with the Agile movement in software development, which includes a reaction against "*big design up front*"—the traditional software engineering practice whereby, after a lengthy requirements-gathering exercise, there is an equally lengthy design stage where the software is planned out on paper. The Agile philosophy is that you learn more from solving problems in practice than in theory, especially when you confront your application with real users as soon as possible. Instead of a long up-front design phase, we try to put a *minimum viable product* out there early, and let the design evolve gradually based on feedback from real-world usage.

But that doesn't mean that thinking about design is outright banned! In Chapter 5, we saw how just blundering ahead without thinking can *eventually* get us to the right answer, but often a little thinking about design can help us get there faster. So, let's think about our minimum viable lists app, and what kind of design we'll need to deliver it:

- We want each user to be able to store their own list—at least one, for now.
- A list is made up of several items, whose primary attribute is a bit of descriptive text.
- We need to save lists from one visit to the next. For now, we can give each user a unique URL for their list. Later on, we may want some way of automatically recognising users and showing them their lists.

To deliver the "for now" items, we're going to have to store lists and their items in a database. Each list will have a unique URL, and each list item will be a bit of descriptive text, associated with a particular list—something like Figure 7-1.

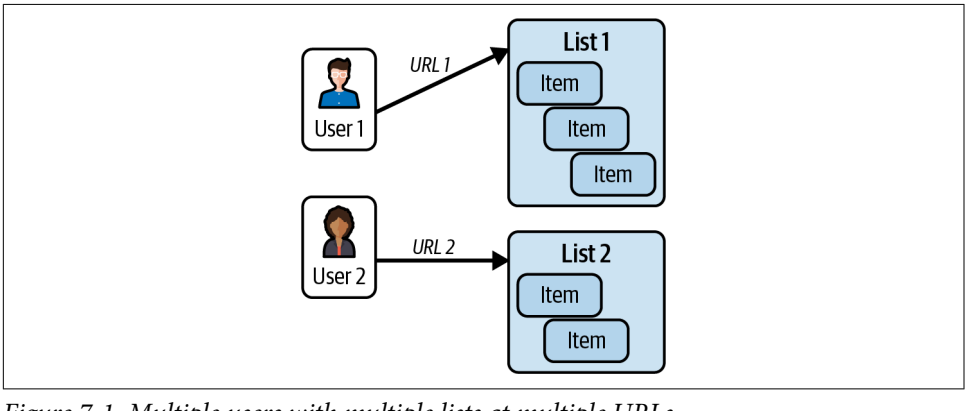


Figure 7-1. Multiple users with multiple lists at multiple URLs

YAGNI!

Once you start thinking about design, it can be hard to stop. All sorts of other thoughts are occurring to us—we might want to give each list a name or title, we might want to recognise users using usernames and passwords, we might want to add a longer notes field as well as short descriptions to our list, we might want to store some kind of ordering, and so on. But we should obey another tenet of the Agile gospel: YAGNI (pronounced yag-knee), which stands for "You ain't gonna need it!" As software developers, we have fun creating, and sometimes it's hard to resist the urge to build things just because an idea occurred to us and we *might* need it. The trouble is that more often than not, no matter how cool the idea was, you *won't*

end up using it. Instead you just end up with a load of unused code, adding to the complexity of your application.

YAGNI is the motto we use to resist our overenthusiastic creative urges. We avoid writing any code that's not strictly required.

Don't write any code unless you absolutely have to.[1]

REST-ish

We have an idea of the data structure we want—the "model" part of model-view-controller (we talked about MVC in "Django's MVC, URLs, and View Functions" on page 29). What about the "view" and "controller" parts? How should the user interact with `Lists` and their `Items` using a web browser?

Representational state transfer (REST) is an approach to web design that's usually used to guide the design of web-based APIs. When designing a user-facing site, it's not possible to stick *strictly* to the REST rules, but they still provide some useful inspiration (take a look at Online Appendix: Building a REST API if you want to see a real REST API). REST suggests that we have a URL structure that matches our data structure—in this case, lists and list items. Each list can have its own URL:

```
/lists/<list identifier>/
```

To view a list, we use a GET request (a normal browser visit to the page).

To create a brand new list, we'll have a special URL that accepts POST requests:

```
/lists/new
```

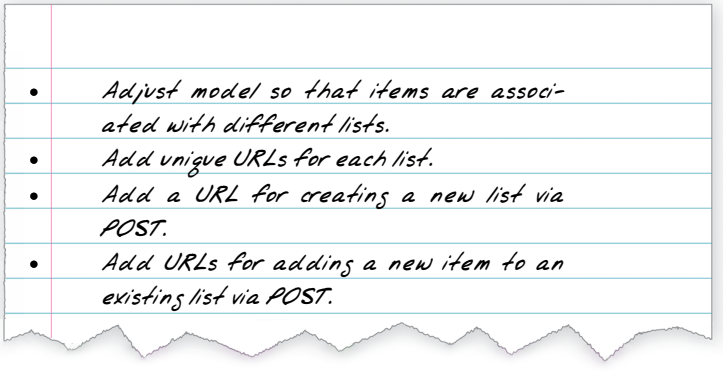
To add a new item to an existing list, we'll have a separate URL, to which we can send POST requests:

```
/lists/<list identifier>/add_item
```

(Again, we're not trying to perfectly follow the rules of REST, which would use a PUT request here—we're just using REST for inspiration. Apart from anything else, you can't use PUT in a standard HTML form.)

1 This is a much more widely applicable rule for programming in business, actually. If you can solve a problem without any coding at all, that's a big win.

In summary, our scratchpad for this chapter looks something like this:



Implementing the New Design Incrementally Using TDD

How do we use TDD to implement the new design? Let's take another look at the flowchart for the TDD process, duplicated in Figure 7-2 for your convenience.

At the top level, we're going to use a combination of adding new functionality (by adding a new FT and writing new application code) and refactoring our application—that is, rewriting some of the existing implementation so that it delivers the same functionality to the user but using aspects of our new design. We'll be able to use the existing FT to verify that we don't break what already works, and the new FT to drive the new features.

At the unit test level, we'll be adding new tests or modifying existing ones to test for the changes we want, and we'll be able to similarly use the unit tests we *don't* touch to help make sure we don't break anything in the process.

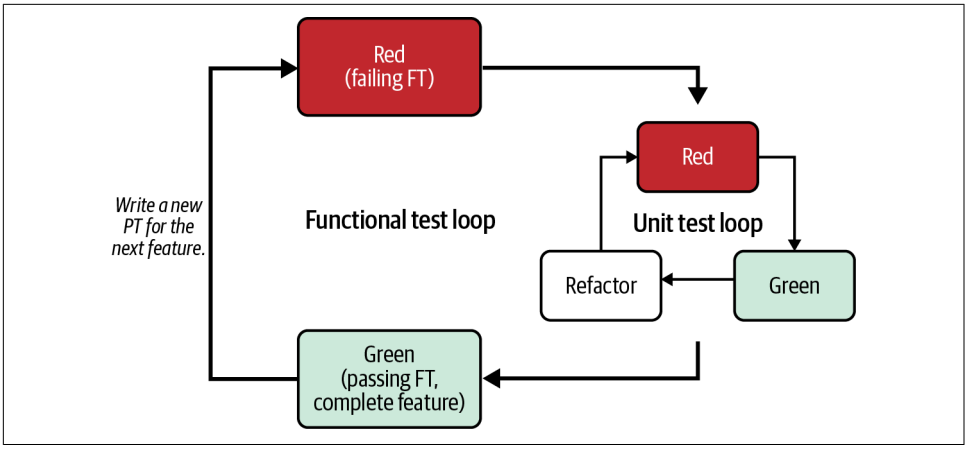


Figure 7-2. The TDD process with both functional and unit tests

Ensuring We Have a Regression Test

Our existing FT, `test_can_start_a_todo_list()`, is going to act as our regression test.

Let's translate our scratchpad into a new FT method, which introduces a second user and checks that their to-do list is separate from Edith's.

We'll start out very similarly to the first. Edith adds a first item to create a to-do list, but we introduce our first new assertion—Edith's list should live at its own, unique URL:

functional_tests/tests.py (ch07l005)

```
def test_can_start_a_todo_list(self):
    # Edith has heard about a cool new online to-do app.
    [...]
    # Satisfied, she goes back to sleep

def test_multiple_users_can_start_lists_at_different_urls(self):
    # Edith starts a new to-do list
    self.browser.get(self.live_server_url)
    inputbox = self.browser.find_element(By.ID, "id_new_item")
    inputbox.send_keys("Buy peacock feathers")
    inputbox.send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table("1: Buy peacock feathers")

    # She notices that her list has a unique URL
    edith_list_url = self.browser.current_url
    self.assertRegex(edith_list_url, "/lists/.+")  ❶
```

❶ `assertRegex` is a helper function from `unittest` that checks whether a string matches a regular expression. We use it to check that our new REST-ish design has been implemented. Find out more in the `unittest` documentation.

Next, we imagine a new user coming along. We want to check that they don't see any of Edith's items when they visit the home page, and that they get their own unique URL for their list:

functional_tests/tests.py (ch07l006)

```
[...]
self.assertRegex(edith_list_url, "/lists/.+")

# Now a new user, Francis, comes along to the site.

## We delete all the browser's cookies
## as a way of simulating a brand new user session  ❶
self.browser.delete_all_cookies()

# Francis visits the home page.  There is no sign of Edith's
# list
self.browser.get(self.live_server_url)
page_text = self.browser.find_element(By.TAG_NAME, "body").text
self.assertNotIn("Buy peacock feathers", page_text)

# Francis starts a new list by entering a new item. He
# is less interesting than Edith...
inputbox = self.browser.find_element(By.ID, "id_new_item")
inputbox.send_keys("Buy milk")
inputbox.send_keys(Keys.ENTER)
self.wait_for_row_in_list_table("1: Buy milk")

# Francis gets his own unique URL
francis_list_url = self.browser.current_url
self.assertRegex(francis_list_url, "/lists/.+")
self.assertNotEqual(francis_list_url, edith_list_url)

# Again, there is no trace of Edith's list
page_text = self.browser.find_element(By.TAG_NAME, "body").text
self.assertNotIn("Buy peacock feathers", page_text)
self.assertIn("Buy milk", page_text)

# Satisfied, they both go back to sleep
```

❶ I'm using the convention of double-hashes (`##`) to indicate "meta-comments"—comments about *how* the test is working and why—so that we can distinguish them from regular comments in FTs, which explain the user story. They're a

message to our future selves, which might otherwise be wondering why we're faffing about deleting cookies…

Other than that, the new test is fairly self-explanatory. Let's see how we do when we run our FTs:

```
$ python manage.py test functional_tests
[...]
.F
======================================================================
FAIL: test_multiple_users_can_start_lists_at_different_urls (functional_tests.t
ests.NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls)

 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 77, in
test_multiple_users_can_start_lists_at_different_urls
    self.assertRegex(edith_list_url, "/lists/.+")
AssertionError: Regex didn't match: '/lists/.+' not found in
'http://localhost:8081/'

 ----------------------------------------------------------------------
Ran 2 tests in 5.786s

FAILED (failures=1)
```

Good, our first test still passes, and the second one fails where we might expect. Let's do a commit, and then go and build some new models and views:

```
$ git commit -a
```

Iterating Towards the New Design

Being all excited about our new design, I had an overwhelming urge to dive in at this point and start changing *models.py*, which would have broken half the unit tests, and then pile in and change almost every single line of code, all in one go. That's a natural urge, and TDD, as a discipline, is a constant fight against it. Obey the Testing Goat, not Refactoring Cat! We don't need to implement our new, shiny design in a single big bang. Let's make small changes that take us from a working state to a working state, with our design guiding us gently at each stage.

There are four items on our to-do list. The FT, with its `Regex didn't match` error, is suggesting to us that the second item—giving lists their own URL and identifier—is the one we should work on next. Let's have a go at fixing that, and only that.

The URL comes from the redirect after POST. In *lists/tests.py*, let's find `test_redirects_after_POST` and change the expected redirect location:

lists/tests.py (ch07l007)

```
def test_redirects_after_POST(self):
    response = self.client.post("/", data={"item_text": "A new list item"})
    self.assertRedirects(response, "/lists/the-only-list-in-the-world/")
```

Does that seem slightly strange? Clearly, */lists/the-only-list-in-the-world* isn't a URL that's going to feature in the final design of our application. But we're committed to changing one thing at a time. While our application only supports one list, this is the only URL that makes sense. We're still moving forwards, in that we'll have a different URL for our list and our home page, which is a step along the way to a more REST-ful design. Later, when we have multiple lists, it will be easy to change.

Another way of thinking about it is as a problem-solving technique: our new URL design is currently not implemented, so it works for zero items. Ultimately, we want to solve for *n* items, but solving for one item is a good step along the way.

Running the unit tests gives us an expected fail:

```
$ python manage.py test lists
[...]
AssertionError: '/' != '/lists/the-only-list-in-the-world/'
- /
+ /lists/the-only-list-in-the-world/
 : Response redirected to '/', expected '/lists/the-only-list-in-the-world/':
Expected '/' to equal '/lists/the-only-list-in-the-world/'.
```

We can go adjust our `home_page` view in *lists/views.py*:

lists/views.py (ch07l008)

```
def home_page(request):
    if request.method == "POST":
        Item.objects.create(text=request.POST["item_text"])
        return redirect("/lists/the-only-list-in-the-world/")

    items = Item.objects.all()
    return render(request, "home.html", {"items": items})
```

Django's unit test runner picks up on the fact that this is not a real URL yet:

```
$ python manage.py test lists
[...]
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
```

Taking a First, Self-Contained Step: One New URL

Our singleton list URL doesn't exist yet. We fix that in *superlists/urls.py*:

superlists/urls.py (ch07l009)

```
from django.urls import path
from lists import views

urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/the-only-list-in-the-world/", views.home_page, name="view_list"),  ❶
]
```

❶ We'll just point our new URL at the existing home page view. This is the minimal change.

Watch out for trailing slashes in URLs, both here in *urls.py* and in the tests. They're a common source of confusion: Django will return a 301 redirect rather than a 404 if you try to access a URL that's missing its trailing slash.[2]

That gets our unit tests passing:

```
$ python manage.py test lists
[...]
OK
```

What do the FTs think?

```
$ python manage.py test functional_tests
[...]
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
list\n1: Buy peacock feathers'
```

Good, they get a little further along. We now confirm that we have a new URL, but the actual page content is still the same; it shows the old list.

Separating Out Our Home Page and List View Functionality

We now have two URLs, but they're actually doing the exact same thing. Under the hood, they're just pointing at the same function. Continuing to work incrementally, we can start to break apart the responsibilities for these two different URLs:

- The home page only needs to display a static form, and support creating a brand new list based on its first item.

2 The setting that controls this is called APPEND_SLASH.

- The list view page needs to be able to display existing list items and add new items to the list.

Let's split out some tests for our new URL.

Open up *lists/tests.py*, and add a new test class called `ListViewTest`. Then:

1. Copy across the `test_renders_input_form()` test from `HomePageTest` into our new class.
2. Move the method called `test_displays_all_list_items()`.
3. In both, change just the URL that is invoked by `self.client.get()`.
4. We *won't* copy across the `test_uses_home_template()` yet, as we're not quite sure what template we want to use. We'll stick to the tests that check behaviour, rather than implementation.

lists/tests.py (ch07l010)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]
    def test_renders_input_form(self):
        [...]
    def test_can_save_a_POST_request(self):
        [...]
    def test_redirects_after_POST(self):
        [...]


class ListViewTest(TestCase):
    def test_renders_input_form(self):
        response = self.client.get("/lists/the-only-list-in-the-world/")
        self.assertContains(response, '<form method="POST">')
        self.assertContains(response, '<input name="item_text"')

    def test_displays_all_list_items(self):
        Item.objects.create(text="itemey 1")
        Item.objects.create(text="itemey 2")

        response = self.client.get("/lists/the-only-list-in-the-world/")

        self.assertContains(response, "itemey 1")
        self.assertContains(response, "itemey 2")
```

Let's try running these tests now:

```
$ python manage.py test lists
OK
```

It passes, because the URL is still pointing at the `home_page` view.

Let's make it point at a new view:

superlists/urls.py (ch07l011)

```
from django.urls import path
from lists import views

urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/the-only-list-in-the-world/", views.view_list, name="view_list"),
]
```

That predictably fails because there is no such view function yet:

```
$ python manage.py test lists
[...]
    path("lists/the-only-list-in-the-world/", views.view_list,
name="view_list"),
                                              ^^^^^^^^^^^^^^^
AttributeError: module 'lists.views' has no attribute 'view_list'
```

A new view function

Fair enough. Let's create a placeholder view function in *lists/views.py*:

lists/views.py (ch07l012-0)

```
def view_list(request):
    pass
```

Not quite good enough:

```
ValueError: The view lists.views.view_list didn't return an HttpResponse
object. It returned None instead.
[...]
FAILED (errors=3)
```

Looking for the minimal code change, let's just make the view return our existing *home.html* template, but with nothing in it:

lists/views.py (ch07l012-1)

```
def view_list(request):
    return render(request, "home.html")
```

Now the tests guide us to making sure that our list view shows existing list items:

```
FAIL: test_displays_all_list_items
(lists.tests.ListViewTest.test_displays_all_list_items)
[...]
AssertionError: False is not true : Couldn't find 'itemey 1' in the following
response
```

So let's copy the last two lines from `home_page` more directly:

lists/views.py (ch07l012)

```
def view_list(request):
    items = Item.objects.all()
    return render(request, "home.html", {"items": items})
```

That gets us to passing unit tests!

```
Ran 8 tests in 0.035s

OK
```

The FTs Detect a Regression

As always when we get to passing unit tests, we run the FTs to check how things are doing "in real life":

```
$ python manage.py test functional_tests
[...]
FF
======================================================================
FAIL: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 62, in
test_can_start_a_todo_list
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']

======================================================================
FAIL: test_multiple_users_can_start_lists_at_different_urls (functional_tests.t
ests.NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 89, in
test_multiple_users_can_start_lists_at_different_urls
    self.assertNotIn("Buy peacock feathers", page_text)
AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
list\n1: Buy peacock feathers'
```

Another Race Condition Example

You may have noticed that the assertions around line 63 are in a slightly unexpected order:

functional_tests/tests.py

```
        # The page updates again, and now shows both items on her list
        self.wait_for_row_in_list_table("2: Use peacock feathers to make a fly")
        self.wait_for_row_in_list_table("1: Buy peacock feathers")
```

Try putting them the other way around, 1 then 2, and run the FTs a few times. There's a good chance you'll notice an inconsistency in the results. Sometimes you see:

```
AssertionError: '1: Buy peacock feathers' not found in ['1: Use peacock
feathers to make a fly']
```

And sometimes you'll see:

```
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
```

That's because of a race condition between the Selenium assertions in the FT, and the server returning our new page. Just before we tap Enter, the page is still showing `1: Buy peacock feathers`. Our next assertion is then checking for `1: Buy peacock feathers`, which is already on the page. But, at the same time, the server is busy returning a new page that also says `1: Use peacock feathers to make a fly`.

So, depending on who gets there first, the first assert may pass or fail, meaning that you may get an error on the first assert or on the second.

That's why I put the assertions "backwards", so we check for `2: Use peacock feathers` *first*, because it should *never* be present on the old page. This means that as soon as we detect it, we must be on the new page.

Subtle, right? Selenium tests are fiddly like that.

Not only is our new FT failing, but the old one is too. That tells us we've introduced a *regression*. But what?

Both tests are failing when we try to add the second item. We have to put our debugging hats on here. We know the home page is working, because the test has got all the way down to line 62 in the first FT, so we've at least added a first item. And our unit tests are all passing, so we're pretty sure the URLs and views that we *do* have are doing what they should. Let's have a quick look at those unit tests to see what they tell us:

```
$ grep -E "class|def" lists/tests.py
class HomePageTest(TestCase):
    def test_uses_home_template(self):
    def test_renders_input_form(self):
    def test_can_save_a_POST_request(self):
    def test_redirects_after_POST(self):
    def test_only_saves_items_when_necessary(self):
class ListViewTest(TestCase):
    def test_renders_input_form(self):
    def test_displays_all_list_items(self):
class ItemModelTest(TestCase):
    def test_saving_and_retrieving_items(self):
```

The home page displays the right form and template, and can handle POST requests, and the */only-list-in-the-world/* view knows how to display all items…but it doesn't know how to handle POST requests. Ah, that gives us a clue.

A second clue is the rule of thumb that, when all the unit tests are passing but the FTs aren't, it's often pointing at a problem in code that's not covered by the unit tests—and in a Django app, that's often a template problem.

Have you figured out what the problem is? Why not spend a moment trying to figure it out? Maybe open up the site in your browser, and see where the bug manifests. Perhaps open up the "view source" or browser DevTools and look at the underlying HTML?

The answer is that our *home.html* input form currently doesn't specify an explicit URL to POST to:

lists/templates/home.html

```
<form method="POST">
```

By default, the browser sends the POST data back to the same URL it's currently on. When we're on the home page that works fine, but when we're on our */only-list-in-the-world/* page, it doesn't.

Getting Back to a Working State as Quickly as Possible

Now, we could dive in and add POST request handling to our new view, but that would involve writing a bunch more tests and code, and at this point we'd like to get back to a working state as quickly as possible. Actually the *quickest* thing we can do to get things fixed is to just use the existing home page view, which already works, for all POST requests.

In other words, we've identified a new important part of the behaviour we want from our two views and their templates, which is the URL that the form points to. Let's add

a check for that URL explicitly, in our two tests for each view (I'll use a diff to show the changes, hopefully that makes it nice and clear):

lists/tests.py (ch07l013-1)

```
@@ -10,7 +10,7 @@ class HomePageTest(TestCase):

     def test_renders_input_form(self):
         response = self.client.get("/")
-        self.assertContains(response, '<form method="POST">')
+        self.assertContains(response, '<form method="POST" action="/">')
         self.assertContains(response, '<input name="item_text"')

     def test_can_save_a_POST_request(self):
@@ -31,7 +31,7 @@ class HomePageTest(TestCase):
 class ListViewTest(TestCase):
     def test_renders_input_form(self):
         response = self.client.get("/lists/the-only-list-in-the-world/")
-        self.assertContains(response, '<form method="POST">')
+        self.assertContains(response, '<form method="POST" action="/">')
         self.assertContains(response, '<input name="item_text"')

     def test_displays_all_list_items(self):
```

That gives us two expected failures:

```
======================================================================
FAIL: test_renders_input_form
(lists.tests.HomePageTest.test_renders_input_form)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/lists/tests.py", line 13, in test_renders_input_form
    self.assertContains(response, '<form method="POST" action="/">')
    ~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/">' in the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST">\n      <input
name="item_text" id="id_new_item" placeholder="Enter a to-do item" />\n
<input type="hidden" name="csrfmiddlewaretoken"
value=[...]
</form>\n    <table id="id_list_table">\n      \n    </table>\n
</body>\n</html>\n'

======================================================================
FAIL: test_renders_input_form
(lists.tests.ListViewTest.test_renders_input_form)
[...]
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/">' in the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
[...]
```

And so we can fix it like this—the input form, for now, will always point at the home URL:

lists/templates/home.html (ch07l013-2)

```
        <form method="POST" action="/">
```

Unit test pass:

```
    OK
```

And we should see our FTs get back to a happier place:

```
    FAIL: test_multiple_users_can_start_lists_at_different_urls (functional_tests.t
    ests.NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls)
    [...]
    AssertionError: 'Buy peacock feathers' unexpectedly found in 'Your To-Do
    list\n1: Buy peacock feathers'

    Ran 2 tests in 8.541s
    FAILED (failures=1)
```

Our old FT (the one we're using as a regression test) passes once again, so we know we're back to a working state. The new functionality may not be working yet, but at least the old stuff works as well as it used to.

Green? Refactor

Time for a little tidying up.

In the red/green/refactor dance, our unit tests pass and all our old FTs pass, so we've arrived at green. That means it's time to see if anything needs a refactor.

We now have two views: one for the home page, and one for an individual list. Both are currently using the same template, and passing it all the list items currently in the database. Post requests are only handled by the home page though.

It feels like the responsibilities of our two views are a little tangled up. Let's try and disentangle them.

Another Small Step: A Separate Template for Viewing Lists

As the home page and the list view are now quite distinct pages, they should be using different HTML templates; *home.html* can have the single input box, whereas a new template, *list.html*, can take care of showing the table of existing items.

We held off on copying across `test_uses_home_template()` until now, because we weren't quite sure what we wanted. Now let's add an explicit test to say that this view uses a different template:

lists/tests.py (ch07l014)

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        response = self.client.get("/lists/the-only-list-in-the-world/")
        self.assertTemplateUsed(response, "list.html")

    def test_renders_input_form(self):
        [...]

    def test_displays_all_list_items(self):
        [...]
```

Let's see what it says:

```
AssertionError: False is not true : Template 'list.html' was not a template
used to render the response. Actual template(s) used: home.html
```

Looks about right, let's change the view:

lists/views.py (ch07l015)

```
def view_list(request):
    items = Item.objects.all()
    return render(request, "list.html", {"items": items})
```

But, obviously, that template doesn't exist yet. If we run the unit tests, we get:

```
django.template.exceptions.TemplateDoesNotExist: list.html
[...]
FAILED (errors=4)
```

Let's create a new file at *lists/templates/list.html*:

```
$ touch lists/templates/list.html
```

A blank template, which gives us two errors—good to know the tests are there to make sure we fill it in:

```
$ python manage.py test lists
[...]
======================================================================
FAIL: test_displays_all_list_items
(lists.tests.ListViewTest.test_displays_all_list_items)
 ----------------------------------------------------------------------
[...]
AssertionError: False is not true : Couldn't find 'itemey 1' in the following
response
b''

======================================================================
FAIL: test_renders_input_form
(lists.tests.ListViewTest.test_renders_input_form)
----------------------------------------------------------------------
[...]
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/">' in the following response
[...]
```

The template for an individual list will reuse quite a lot of the stuff we currently have in *home.html*, so we can start by just copying that:

```
$ cp lists/templates/home.html lists/templates/list.html
```

That gets the tests back to passing (green).

```
$ python manage.py test lists
[...]
OK
```

Now let's do a little more tidying up (refactoring). We said the home page doesn't need to list items; it only needs the new list input field. So we can remove some lines from *lists/templates/home.html*, and maybe slightly tweak the `h1` to say "Start a new To-Do list".

I'll present the code change as a diff again, as I think that shows nice and clearly what we need to modify:

lists/templates/home.html (ch07l018)

```
   <body>
-    <h1>Your To-Do list</h1>
+    <h1>Start a new To-Do list</h1>
     <form method="POST" action="/">
       <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
       {% csrf_token %}
     </form>
-    <table id="id_list_table">
-      {% for item in items %}
-        <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
-      {% endfor %}
-    </table>
   </body>
```

We rerun the unit tests to check that hasn't broken anything…

```
OK
```

Good.

Now there's actually no need to pass all the items to the *home.html* template in our `home_page` view, so we can simplify that and delete a few lines:

lists/views.py (ch07l019)

```
    if request.method == "POST":
        Item.objects.create(text=request.POST["item_text"])
        return redirect("/lists/the-only-list-in-the-world/")
-
-   items = Item.objects.all()
-   return render(request, "home.html", {"items": items})
+   return render(request, "home.html")
```

Rerun the unit tests once more; they still pass:

```
OK
```

Time to run the FTs:

```
  File "...goat-book/functional_tests/tests.py", line 96, in
test_multiple_users_can_start_lists_at_different_urls
    self.wait_for_row_in_list_table("1: Buy milk")
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^
[...]
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']

----------------------------------------------------------------------
Ran 2 tests in 10.606s

FAILED (failures=1)
```

Great! Only one failure, so we know our regression test (the first FT) is passing. Let's see where we're getting to with the new FT.

Let's take a look at it again:

functional_tests/tests.py

```
def test_multiple_users_can_start_lists_at_different_urls(self):
    # Edith starts a new to-do list
    self.browser.get(self.live_server_url)
    inputbox = self.browser.find_element(By.ID, "id_new_item")
    inputbox.send_keys("Buy peacock feathers")
    inputbox.send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table("1: Buy peacock feathers")  ❶
    [...]

    # Now a new user, Francis, comes along to the site.
    [...]

    # Francis visits the home page.  There is no sign of Edith's
    # list
    self.browser.get(self.live_server_url)
    page_text = self.browser.find_element(By.TAG_NAME, "body").text
    self.assertNotIn("Buy peacock feathers", page_text)  ❷

    # Francis starts a new list by entering a new item. He
    # is less interesting than Edith...
    inputbox = self.browser.find_element(By.ID, "id_new_item")
    inputbox.send_keys("Buy milk")
    inputbox.send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table("1: Buy milk")   ❸
    [...]
```

❶ Edith's list says "Buy peacock feathers".

❷ When Francis loads the home page, there's no sign of Edith's list.

❸ (This is the line where our test fails.) When Francis adds a new item, he sees Edith's item as number 1, and his appears as number 2.

Still, that's progress! The new FT *is* getting a little further along.

It may feel like we haven't made much headway because, functionally, the site still behaves almost exactly like it did when we started the chapter. But this really *is* progress. We've started on the road to our new design, and we've implemented a number of stepping stones *without making anything worse than it was before*.

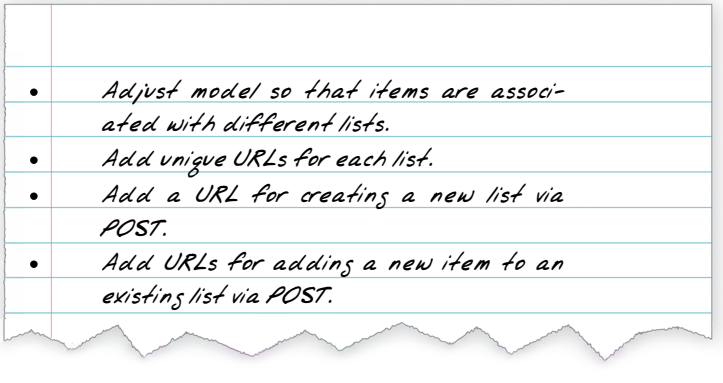
Let's commit our work so far:

```
$ git status # should show 4 changed files and 1 new file, list.html
$ git add lists/templates/list.html
$ git diff # should show we've simplified home.html,
           # moved one test to a new class in lists/tests.py,
           # changed the redirect in homepageTest & the home_page() view
           # added a new view view_list() in views.py,
           # and and added a line to urls.py.
$ git commit -a # add a message summarising the above, maybe something like
                # "new URL, view and template to display lists"
```

If this is all feeling a little abstract, now might be a good time to load up the site with `manage.py runserver` and try adding a couple of different lists yourself, and get a feel for how the site is currently behaving.

A Third Small Step: A New URL for Adding List Items

Where are we with our own to-do list?



We've *sort of* made progress on the second item, even if there's still only one list in the world. The first item is a bit scary. Can we do something about items 3 or 4?

Let's have a new URL for adding new list items at */lists/new*: If nothing else, it'll simplify the home page view.

A Test Class for New List Creation

Open up *lists/tests.py*, and *move* the `test_can_save_a_POST_request()` and `test_redirects_after_POST()` methods into a new class called `NewListTest`. Then, change the URL they POST to:

lists/tests.py (ch07l020)

```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]
    def test_renders_input_form(self):
        [...]
    def test_only_saves_items_when_necessary(self):
        [...]


class NewListTest(TestCase):
    def test_can_save_a_POST_request(self):
        self.client.post("/lists/new", data={"item_text": "A new list item"})
        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.get()
        self.assertEqual(new_item.text, "A new list item")

    def test_redirects_after_POST(self):
        response = self.client.post("/lists/new", data={"item_text": "A new list item"})
        self.assertRedirects(response, "/lists/the-only-list-in-the-world/")


class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]
```

This is another place to pay attention to trailing slashes, incidentally. It's `/lists/new`, with no trailing slash. The convention I'm using is that URLs without a trailing slash are "action" URLs, which modify the database.[3]

Try running that:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
[...]
    self.assertRedirects(response, "/lists/the-only-list-in-the-world/")
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

3 I don't think this is a very common convention anymore these days, but I quite like it. By all means, cast around for a URL naming scheme that makes sense to you in your own projects!

The first failure tells us we're not saving a new item to the database, and the second says that, instead of returning a 302 redirect, our view is returning a 404. That's because we haven't built a URL for */lists/new*, so the `client.post` is just getting a "not found" response.

Do you remember how we split this out into two tests earlier? If we only had one test that checked both the saving and the redirect, it would have failed on the `0 != 1` failure, which would have been much harder to debug. Ask me how I know this.

A URL and View for New List Creation

Let's build our new URL now:

superlists/urls.py (ch07l021)

```
urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/new", views.new_list, name="new_list"),
    path("lists/the-only-list-in-the-world/", views.view_list, name="view_list"),
]
```

Next we get a `no attribute 'new_list'`, so let's fix that, in *lists/views.py*:

lists/views.py (ch07l022)

```
def new_list(request):
    pass
```

Then we get "The view lists.views.new_list didn't return an HttpResponse object". (This is getting rather familiar!) We could return a raw `HttpResponse`, but because we know we'll need a redirect, let's borrow a line from `home_page`:

lists/views.py (ch07l023)

```
def new_list(request):
    return redirect("/lists/the-only-list-in-the-world/")
```

That gives:

```
    self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Seems reasonably straightforward. We borrow another line from `home_page`:

lists/views.py (ch07l024)

```
def new_list(request):
    Item.objects.create(text=request.POST["item_text"])
    return redirect("/lists/the-only-list-in-the-world/")
```

And everything now passes:

```
Ran 9 tests in 0.030s

OK
```

And we can run the FTs to check that we're still in the same place:

```
[...]
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
Ran 2 tests in 8.972s
FAILED (failures=1)
```

Our regression test passes, and the new FT gets to the same point.

Removing Now-Redundant Code and Tests

We're looking good. As our new views are now doing most of the work that `home_page` used to do, we should be able to massively simplify it. Can we remove the whole `if request.method == 'POST'` section, for example?

lists/views.py (ch07l025)

```
def home_page(request):
    return render(request, "home.html")
```

Yep! The unit tests pass:

```
OK
```

And while we're at it, we can remove the now-redundant `test_only_saves_items_when_necessary` test too!

Doesn't that feel good? The view functions are looking much simpler. We rerun the tests to make sure…

```
Ran 8 tests in 0.016s
OK
```

And the FTs?

A Regression! Pointing Our Forms at the New URL

Oops. When we run the FTs:

```
======================================================================
ERROR: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
[...]
  File "...goat-book/functional_tests/tests.py", line 52, in
test_can_start_a_todo_list
[...]
    self.wait_for_row_in_list_table("1: Buy peacock feathers")
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^
[...]
    table = self.browser.find_element(By.ID, "id_list_table")
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; For documentation [...]


======================================================================
ERROR: test_multiple_users_can_start_lists_at_different_urls (functional_tests.
tests.NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls)
 ----------------------------------------------------------------------
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; For documentation [...]
[...]

Ran 2 tests in 11.592s
FAILED (errors=2)
```

Once again, the FTs pick up a tricky little bug, something that our unit tests alone would find it hard to catch.

Debugging in DevTools

This is another good time to spin up the dev server, and have a look around with a browser.

Let's also open up DevTools,[4] and click around to see what's going on:

- First I tried submitting a new list item, and saw we get sent back to the home page.
- Then I did the same with the browser DevTools open, and in the "network" tab I saw a POST request to "/". See Figure 7-3.
- Finally, I had a look at the HTML source of the home page, and saw that the main form is still pointing at "/".

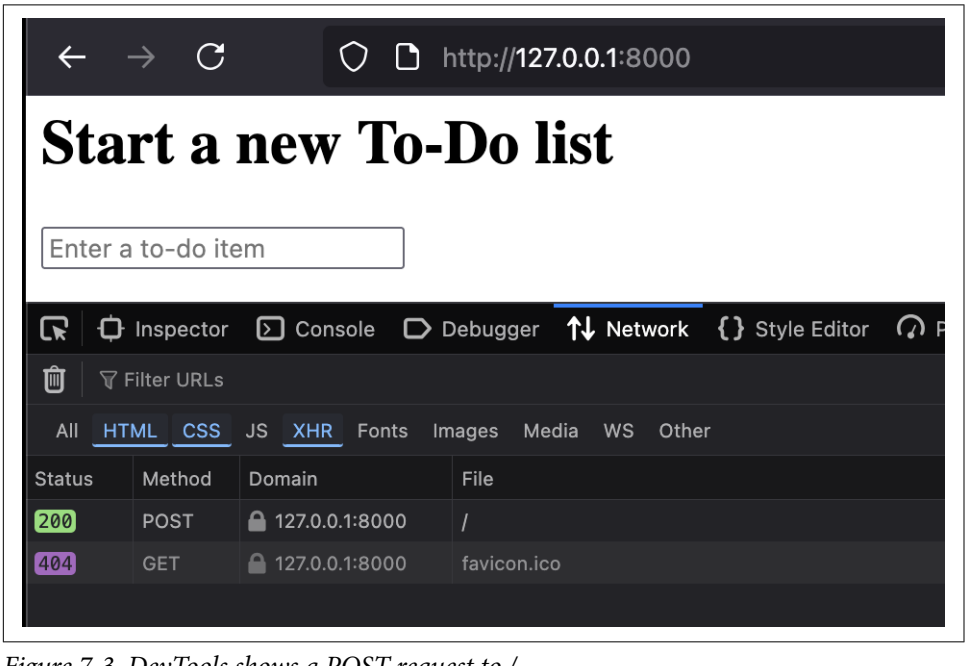


Figure 7-3. DevTools shows a POST request to /

4 If you've not seen it before, DevTools is short for "developer tools". They're tools that Firefox (and other browsers) give you to be able to look "under the hood" and see what's going on with web pages, including the source code, what network requests are being made, and what JavaScript is doing. You can open up DevTools with Ctrl+Shift+I or Cmd-Opt-I.

Actually, *both* our forms are still pointing to the old URL. We have tests for this! Let's amend them:

lists/tests.py (ch07l026)

```
@@ -10,7 +10,7 @@ class HomePageTest(TestCase):

     def test_renders_input_form(self):
         response = self.client.get("/")
-        self.assertContains(response, '<form method="POST" action="/">')
+        self.assertContains(response, '<form method="POST" action="/lists/new">')
         self.assertContains(response, '<input name="item_text"')


@@ -33,7 +33,7 @@ class ListViewTest(TestCase):

     def test_renders_input_form(self):
         response = self.client.get("/lists/the-only-list-in-the-world/")
-        self.assertContains(response, '<form method="POST" action="/">')
+        self.assertContains(response, '<form method="POST" action="/lists/new">')
         self.assertContains(response, '<input name="item_text"')
```

That gets us two failures:

```
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/new">' in the following response
[...]
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/new">' in the following response
[...]
```

In *both home.html* and *list.html*, let's change them:

lists/templates/home.html (ch07l027)

```
    <form method="POST" action="/lists/new">
```

And:

lists/templates/list.html (ch07l028)

```
    <form method="POST" action="/lists/new">
```

And that should get us back to working again:

```
Ran 8 tests in 0.006s

OK
```

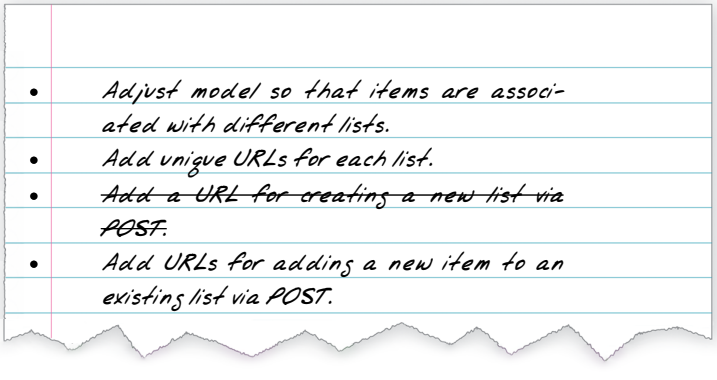
And our FTs are still in the familiar "Francis sees Edith's items" place:

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
[...]
FAILED (failures=1)
```

Perhaps this all seems quite pernickety, but that's another nicely self-contained commit, in that we've made a bunch of changes to our URLs, our *views.py* is looking much neater and tidier with three very short view functions, and we're sure the application is still working as well as it was before. We're getting good at this working-state-to-working-state malarkey!

```
$ git status # 5 changed files
$ git diff # URLs for forms x2, new test + view with code moves, and new URL
$ git commit -a
```

And we can cross out an item on the to-do list:



Biting the Bullet: Adjusting Our Models

Enough housekeeping with our URLs. It's time to bite the bullet and change our models. Let's adjust the model unit test.

lists/tests.py (ch07l029)

```
@@ -1,5 +1,5 @@
 from django.test import TestCase
-from lists.models import Item
+from lists.models import Item, List


 class HomePageTest(TestCase):
@@ -35,20 +35,30 @@ class ListViewTest(TestCase):
         self.assertContains(response, "itemey 2")


-class ItemModelTest(TestCase):
+class ListAndItemModelsTest(TestCase):
     def test_saving_and_retrieving_items(self):
+        mylist = List()
+        mylist.save()
+
         first_item = Item()
         first_item.text = "The first (ever) list item"
+        first_item.list = mylist
         first_item.save()

         second_item = Item()
         second_item.text = "Item the second"
+        second_item.list = mylist
         second_item.save()

+        saved_list = List.objects.get()
+        self.assertEqual(saved_list, mylist)
+
         saved_items = Item.objects.all()
         self.assertEqual(saved_items.count(), 2)

         first_saved_item = saved_items[0]
         second_saved_item = saved_items[1]
         self.assertEqual(first_saved_item.text, "The first (ever) list item")
+        self.assertEqual(first_saved_item.list, mylist)
         self.assertEqual(second_saved_item.text, "Item the second")
+        self.assertEqual(second_saved_item.list, mylist)
```

Once again, this is a very verbose test, because I'm using it more as a demonstration of how the ORM works. We'll shorten it later,[5] but for now, let's work through and see how things work.

We create a new `List` object and then we assign each item to it by setting it as its `.list` property. We check that the list is properly saved, and we check that the two items have also saved their relationship to the list. You'll also notice that we can compare list objects with each other directly (`saved_list` and `mylist`)—behind the scenes, these will compare themselves by checking that their primary key (the `.id` attribute) is the same.

Time for another unit-test/code cycle.

For the first few iterations, rather than explicitly showing you what code to enter in between every test run, I'm only going to show you the expected error messages from running the tests. I'll let you figure out what each minimal code change should be, on your own.

Need a hint? Go back and take a look at the steps we took to introduce the `Item` model in "The Django ORM and Our First Model" on page 81.

Your first error should be:

```
ImportError: cannot import name 'List' from 'lists.models'
```

Fix that, and then you should see:

```
AttributeError: 'List' object has no attribute 'save'
```

Next you should see:

```
django.db.utils.OperationalError: no such table: lists_list
```

So, we run a `makemigrations`:

```
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0003_list.py
    + Create model List
```

And then you should see:

```
    self.assertEqual(first_saved_item.list, mylist)
AttributeError: 'Item' object has no attribute 'list'
```

5 In "Rewriting the Old Model Test" on page 367, if you're curious.

A Foreign Key Relationship

How do we give our `Item` a list attribute? Let's just try naively making it like the `text` attribute (and here's your chance to see whether your solution so far looks like mine, by the way):

lists/models.py (ch07l033)

```
from django.db import models


class List(models.Model):
    pass


class Item(models.Model):
    text = models.TextField(default="")
    list = models.TextField(default="")
```

As usual, the tests tell us we need a migration:

```
$ python manage.py test lists
[...]
django.db.utils.OperationalError: no such column: lists_item.list

$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0004_item_list.py
    + Add field list to item
```

Let's see what that gives us:

```
AssertionError: 'List object (1)' != <List: List object (1)>
```

We're not quite there. Look closely at each side of the `!=`. Do you see the quotes (`'`)? Django has only saved the string representation of the `List` object. To save the relationship to the object itself, we tell Django about the relationship between the two classes using a `ForeignKey`:

lists/models.py (ch07l035)

```
class Item(models.Model):
    text = models.TextField(default="")
    list = models.ForeignKey(List, default=None, on_delete=models.CASCADE)
```

That'll need a migration too. As the last one was a red herring, let's delete it and replace it with a new one:

```
$ rm lists/migrations/0004_item_list.py
$ python manage.py makemigrations
Migrations for 'lists':
  lists/migrations/0004_item_list.py
    + Add field list to item
```

Deleting migrations is dangerous. Now and again it's nice to do it to keep things tidy, because we don't always get our models' code right on the first go! But if you delete a migration that's already been applied to a database somewhere, Django will be confused about what state it's in, and won't be able to apply future migrations. You should only do it when you're sure the migration hasn't been used. A good rule of thumb is that you should never delete or modify a migration that's already been committed to Git.

Adjusting the Rest of the World to Our New Models

Back in our tests, now what happens?

```
$ python manage.py test lists
[...]
ERROR: test_displays_all_list_items
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_redirects_after_POST
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
[...]
ERROR: test_can_save_a_POST_request
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id

Ran 8 tests in 0.021s

FAILED (errors=3)
```

Oh dear!

There is some good news. Although it's hard to see, our model tests are passing. But three of our view tests are failing nastily.

The cause is the new relationship we've introduced between `Items` and `Lists`, which requires each item to have a parent list, and which our old tests and code aren't prepared for.

Still, this is exactly why we have tests! Let's get them working again. The easiest is the `ListViewTest`; we just create a parent list for our two test items:

lists/tests.py (ch07l038)

```
class ListViewTest(TestCase):
    [...]
    def test_displays_all_list_items(self):
        mylist = List.objects.create()
        Item.objects.create(text="itemey 1", list=mylist)
        Item.objects.create(text="itemey 2", list=mylist)
```

That gets us down to two failing tests, both on tests that try to POST to our `new_list` view. Decode the tracebacks using our usual technique, working back from error to line of test code to—buried in there somewhere—the line of our own code that caused the failure:

```
  File "...goat-book/lists/tests.py", line 25, in test_redirects_after_POST
    response = self.client.post("/lists/new", data={"item_text": "A new list
item"})
[...]
  File "...goat-book/lists/views.py", line 11, in new_list
    Item.objects.create(text=request.POST["item_text"])
    ~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[...]
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

It's when we try to create an item without a parent list. So we make a similar change in the view:

lists/views.py (ch07l039)

```
from lists.models import Item, List
[...]

def new_list(request):
    nulist = List.objects.create()
    Item.objects.create(text=request.POST["item_text"], list=nulist)
    return redirect("/lists/the-only-list-in-the-world/")
```

And that gets our tests passing again:[6]

```
Ran 8 tests in 0.030s

OK
```

6 Are you wondering about the strange spelling of the "nulist" variable? Other options are "list", which would shadow the built-in `list()` function, and `new_list`, which would shadow the name of the function that contains it. Or `list_` with the trailing underscore, which I find a bit ugly, or `list1` or `listey` or `mylist`, but none are particularly satisfactory.

Are you cringing internally at this point? *Arg! This feels so wrong; we create a new list for every single new item submission, and we're still just displaying all items as if they belong to the same list!* I know; I feel the same. The step-by-step approach, in which you go from working code to working code, is counterintuitive. I always feel like just diving in and trying to fix everything all in one go, instead of going from one weird half-finished state to another. But remember the Testing Goat! When you're up a mountain, you want to think very carefully about where you put each foot, and take one step at a time, checking at each stage that the place you've put it hasn't caused you to fall off a cliff.

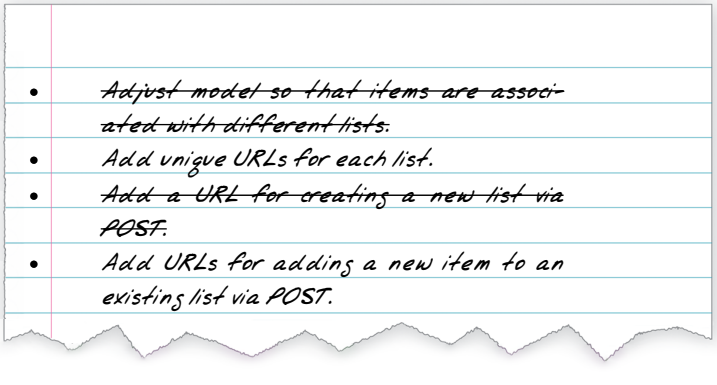
So, just to reassure ourselves that things have worked, we rerun the FT:

```
AssertionError: '1: Buy milk' not found in ['1: Buy peacock feathers', '2: Buy
milk']
[...]
```

Sure enough, it gets all the way through to where we were before. We haven't broken anything, and we've made a big change to the database. That's something to be pleased with! Let's commit:

```
$ git status # 3 changed files, plus 2 migrations
$ git add lists
$ git diff --staged
$ git commit
```

And we can cross out another item on the to-do list:



Each List Should Have Its Own URL

We can get rid of the silly `the-only-list-in-the-world` URL, but what shall we use as the unique identifier for our lists? Probably the simplest thing, for now, is just to use the autogenerated `id` field from the database. Let's change `ListViewTest` so that the two tests point at new URLs.

We'll also change the old `test_displays_all_list_items` test and call it `test_displays_only_items_for_that_list` instead, making it check that only the items for a specific list are displayed:

lists/tests.py (ch07l040)

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")  ❶
        self.assertTemplateUsed(response, "list.html")

    def test_renders_input_form(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")  ❶
        self.assertContains(response, '<form method="POST" action="/lists/new">')
        self.assertContains(response, '<input name="item_text"')

    def test_displays_only_items_for_that_list(self):
        correct_list = List.objects.create()  ❷
        Item.objects.create(text="itemey 1", list=correct_list)
        Item.objects.create(text="itemey 2", list=correct_list)
        other_list = List.objects.create()  ❷
        Item.objects.create(text="other list item", list=other_list)

        response = self.client.get(f"/lists/{correct_list.id}/")  ❸

        self.assertContains(response, "itemey 1")
        self.assertContains(response, "itemey 2")
        self.assertNotContains(response, "other list item")  ❹
```

❶ Here's where we incorporate the ID of our new list into the GET URL.

❷ In the "Given" phase of the test, we now set up two lists: the one we're interested in and an extraneous one.

❸ We change this URL too, to point at the *correct* list.

❹ And now, our "Then" section can check that the irrelevant list's items are definitely not present.

Running the unit tests gives the expected 404s and another related error:

```
FAIL: test_displays_only_items_for_that_list
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)
[...]
FAIL: test_renders_input_form
AssertionError: 404 != 200 : Couldn't retrieve content: Response code was 404
(expected 200)
[...]
FAIL: test_uses_list_template
AssertionError: No templates used to render the response
```

Capturing Parameters from URLs

It's time to learn how we can pass parameters from URLs to views:

superlists/urls.py (ch07l041-0)

```
urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/new", views.new_list, name="new_list"),
    path("lists/<int:list_id>/", views.view_list, name="view_list"),
]
```

We adjust the path string for our URL to include a *capture group*, `<int:list_id>`, which will match any numerical characters, up to the following `/`. The captured `id` will be passed to the view as an argument.

In other words, if we go to the URL */lists/1/*, `view_list` will get a second argument after the normal `request` argument, namely the integer `1`.

But our view doesn't expect an argument yet! Sure enough, this causes problems:

```
ERROR: test_displays_only_items_for_that_list
[...]
TypeError: view_list() got an unexpected keyword argument 'list_id'
[...]
ERROR: test_renders_input_form
[...]
TypeError: view_list() got an unexpected keyword argument 'list_id'
[...]
ERROR: test_uses_list_template
[...]
TypeError: view_list() got an unexpected keyword argument 'list_id'
[...]
FAIL: test_redirects_after_POST
[...]
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
[...]
FAILED (failures=1, errors=3)
```

We can fix that easily with an unused parameter in *views.py*:

lists/views.py (ch07l041)

```
def view_list(request, list_id):
    [...]
```

That takes us down to our expected failure, plus something to do with an */only-list-in-the-world/* that's still hanging around somewhere, which I'm sure we can fix later.

```
FAIL: test_displays_only_items_for_that_list
[...]
AssertionError: 1 != 0 : 'other list item' unexpectedly found in the following
response
[...]
FAIL: test_redirects_after_POST
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
```

Let's make our list view discriminate over which items it sends to the template:

lists/views.py (ch07l042)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    items = Item.objects.filter(list=our_list)
    return render(request, "list.html", {"items": items})
```

Adjusting new_list to the New World

It's time to address the */only-list-in-the-world/* failure:

```
FAIL: test_redirects_after_POST
[...]
AssertionError: 404 != 200 : Couldn't retrieve redirection page
'/lists/the-only-list-in-the-world/': response code was 404 (expected 200)
```

Let's have a little look and find the test that's moaning:

lists/tests.py

```
class NewListTest(TestCase):
    [...]

    def test_redirects_after_POST(self):
        response = self.client.post("/lists/new", data={"item_text": "A new list item"})
        self.assertRedirects(response, "/lists/the-only-list-in-the-world/")
```

It looks like it hasn't been adjusted to the new world of `Lists` and `Items`. The test should be saying that this view redirects to the URL of the specific new list it just created.

lists/tests.py (ch07l043)

```
    def test_redirects_after_POST(self):
        response = self.client.post("/lists/new", data={"item_text": "A new list item"})
        new_list = List.objects.get()
        self.assertRedirects(response, f"/lists/{new_list.id}/")
```

The test still fails, but we can now take a look at the view itself, and change it so it redirects to the right place:

lists/views.py (ch07l044)

```
def new_list(request):
    nulist = List.objects.create()
    Item.objects.create(text=request.POST["item_text"], list=nulist)
    return redirect(f"/lists/{nulist.id}/")
```

That gets us back to passing unit tests, phew!

```
$ python manage.py test lists
[...]
........
 ---------------------------------------------------------------------
Ran 8 tests in 0.033s

OK
```

What about the FTs?

The Functional Tests Detect Another Regression

It feels like we're done with migrating to the new URL structure; we must be almost there?

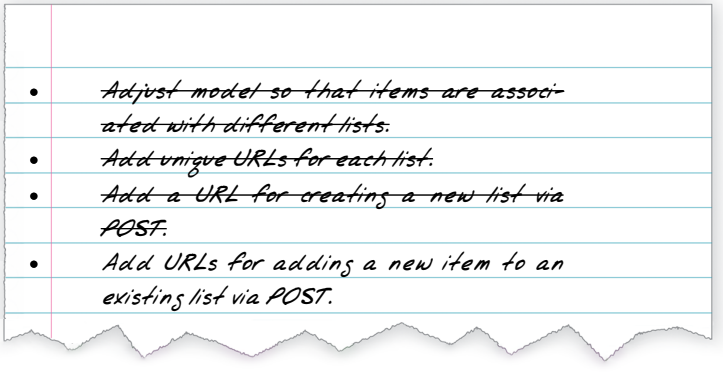
Well, almost. When we run the FTs, we get:

```
F.
======================================================================
FAIL: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ---------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 62, in
test_can_start_a_todo_list
    self.wait_for_row_in_list_table("2: Use peacock feathers to make a fly")
[...]
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Use
peacock feathers to make a fly']

 ---------------------------------------------------------------------
Ran 2 tests in 8.617s

FAILED (failures=1)
```

Our *new* FT is actually passing: different users can get different lists. But the old test is warning us of a regression. It looks like you can't add a second item to a list any more.

It's because of our quick-and-dirty hack where we create a new list for every single POST submission. This is exactly what we have FTs for!

And it correlates nicely with the last item on our to-do list:



One More URL to Handle Adding Items to an Existing List

We need a URL and view to handle adding a new item to an existing list (*/lists/<list_id>/add_item*). We're starting to get used to these now, so we know we'll need:

1. A new test for the new URL
2. A new entry in *urls.py*
3. A new view function

So, let's see if we can knock all that together quickly:

lists/tests.py (ch07l045)

```
class NewItemTest(TestCase):
    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f"/lists/{correct_list.id}/add_item",
            data={"item_text": "A new item for an existing list"},
        )

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.get()
        self.assertEqual(new_item.text, "A new item for an existing list")
        self.assertEqual(new_item.list, correct_list)

    def test_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            f"/lists/{correct_list.id}/add_item",
            data={"item_text": "A new item for an existing list"},
        )

        self.assertRedirects(response, f"/lists/{correct_list.id}/")
```

Are you wondering about `other_list`? A bit like in the tests for viewing a specific list, it's important that we add items to a specific list. Adding this second object to the database prevents me from using a hack like `List.objects.first()` in the view. Yes, that would be a silly thing to do, and you can go too far down the road of testing for all the silly things you must not do (there are an infinite number of those, after all). It's a judgement call, but this one feels worth it. There's some more discussion of this in "An Aside on When to Test for Developer Silliness" on page 366. Oh, and yes it's an unused variable, and your IDE might nag you about it, but I find it helps me to remember what it's for.

So that fails as expected, the list item is not saved, and the new URL currently returns a 404:

```
AssertionError: 0 != 1
[...]
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

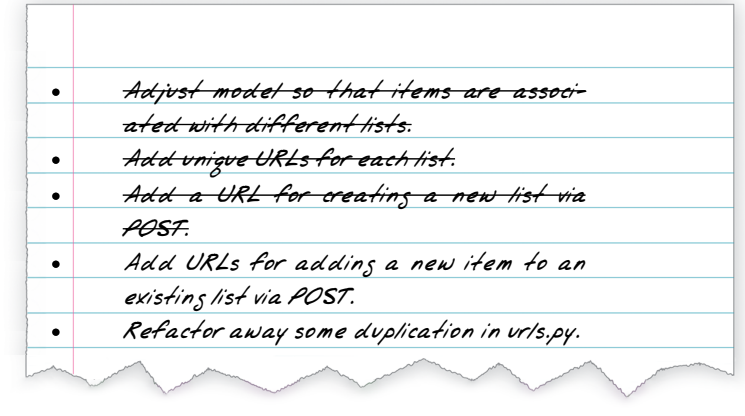
The Last New urls.py Entry

Now we've got our expected 404, let's add a new URL for adding new items to existing lists:

superlists/urls.py (ch07l046)

```
urlpatterns = [
    path("", views.home_page, name="home"),
    path("lists/new", views.new_list, name="new_list"),
    path("lists/<int:list_id>/", views.view_list, name="view_list"),
    path("lists/<int:list_id>/add_item", views.add_item, name="add_item"),
]
```

We've got three very similar-looking URLs there. Let's make a note on our to-do list; they look like good candidates for a refactoring:



The Last New View

Back to the tests, we get the usual missing module view objects:

```
AttributeError: module 'lists.views' has no attribute 'add_item'
```

Let's try:

lists/views.py (ch07l047)

```
def add_item(request):
    pass
```

Aha:

```
TypeError: add_item() got an unexpected keyword argument 'list_id'
```

lists/views.py (ch07l048)

```
def add_item(request, list_id):
    pass
```

And then:

```
ValueError: The view lists.views.add_item didn't return an HttpResponse object.
It returned None instead.
```

We can copy the `redirect()` from `new_list` and the `List.objects.get()` from `view_list`:

lists/views.py (ch07l049)

```
def add_item(request, list_id):
    our_list = List.objects.get(id=list_id)
    return redirect(f"/lists/{our_list.id}/")
```

That takes us to:

```
        self.assertEqual(Item.objects.count(), 1)
AssertionError: 0 != 1
```

Finally, we make it save our new list item:

lists/views.py (ch07l050)

```
def add_item(request, list_id):
    our_list = List.objects.get(id=list_id)
    Item.objects.create(text=request.POST["item_text"], list=our_list)
    return redirect(f"/lists/{our_list.id}/")
```

And we're back to passing tests:

```
Ran 10 tests in 0.050s

OK
```

Hooray! Did that feel like quite a nice, fluid, unit-test/code cycle?

Testing Template Context Directly

We've got our new view and URL for adding items to existing lists; now we just need to actually use it in our *list.html* template. We have a unit test for the form's action; let's amend it:

lists/tests.py (ch07l051)

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]

    def test_renders_input_form(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")
        self.assertContains(
            response,
            f'<form method="POST" action="/lists/{mylist.id}/add_item">',
        )
        self.assertContains(response, '<input name="item_text"')

    def test_displays_only_items_for_that_list(self):
        [...]
```

That fails as expected:

```
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/1/add_item">' in the following response
[...]
```

So, we open it up to adjust the form tag…

lists/templates/list.html

```
    <form method="POST" action="but what should we put here?">
```

…oh.

To get the URL to add to the current list, the template needs to know what list it's rendering, as well as what the items are.

Well, "programming by wishful thinking",[7] let's just pretend we had access to everything we need, like a `list` variable in the template:

lists/templates/list.html (ch07l052)

```
<form method="POST" action="/lists/{{ list.id }}/add_item">
```

That changes our error slightly:

```
AssertionError: False is not true : Couldn't find '<form method="POST"
action="/lists/1/add_item">' in the following response
b'<html>\n  <head>\n    <title>To-Do lists</title>\n  </head>\n  <body>\n
<h1>Your To-Do list</h1>\n    <form method="POST" action="/lists//add_item">\n  ❶
```

❶ Do you see it says `/lists//add_item`? It's because Django templates will just silently ignore any undefined variables, and substitute empty strings for them.

Let's see if we can make our wish come true and pass our list to the template then:

lists/views.py (ch07l053)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    items = Item.objects.filter(list=our_list)
    return render(request, "list.html", {"items": items, "list": our_list})
```

That gets us to passing tests:

```
OK
```

And we now have an opportunity to refactor, as passing both the list and its items together is redundant. Here's the change in the template:

lists/templates/list.html (ch07l054)

```
{% for item in list.item_set.all %}  ❶
  <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
{% endfor %}
```

❶ `.item_set` is called a reverse lookup. It's one of Django's incredibly useful bits of ORM that lets you look up an object's related items from a different table.

The tests still pass…

```
OK
```

7 TDD is a bit like programming by wishful thinking, in that, when we write the tests before the implementation, we express a wish: we wish we had some code that worked! The phrase "programming by wishful thinking" actually has a wider meaning, of writing your code in a top-down kind of way. We'll come back and talk about it more in Chapter 24.

And we can now simplify the view down a little:

lists/views.py (ch07l055)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    return render(request, "list.html", {"list": our_list})
```

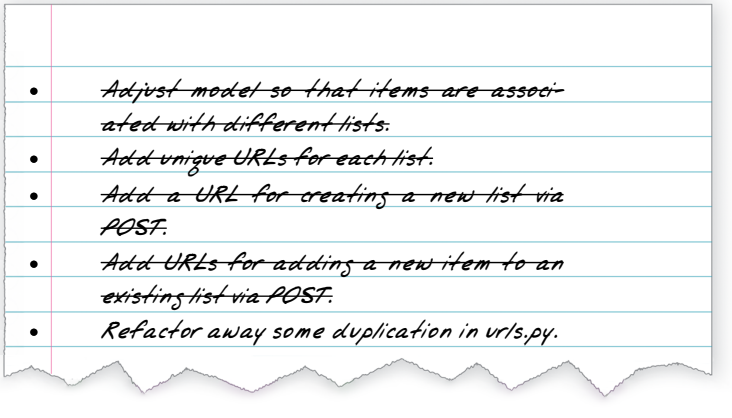
And our unit tests still pass:

```
Ran 10 tests in 0.040s

OK
```

How about the FTs?

```
$ python manage.py test functional_tests
[...]
..
 ----------------------------------------------------------------------
Ran 2 tests in 9.771s

OK
```

HOORAY! Oh, and a quick check on our to-do list:



Irritatingly, the Testing Goat is a stickler for tying up loose ends too, so we've got to do one final thing. Before we start, we'll do a commit—always make sure you've got a commit of a working state before embarking on a refactor:

```
$ git diff
$ git commit -am "new URL + view for adding to existing lists. FT passes :-)"
```

A Final Refactor Using URL includes

superlists/urls.py is really meant for URLs that apply to your entire site. For URLs that only apply to the `lists` app, Django encourages us to use a separate *lists/urls.py*, to make the app more self-contained. The simplest way to make one is to use a copy of the existing *urls.py*:

```
$ cp superlists/urls.py lists/
```

Then we replace the three list-specific lines in *superlists/urls.py* with an `include()`:

superlists/urls.py (ch07l057)

```
from django.urls import include, path
from lists import views as list_views  ❶

urlpatterns = [
    path("", list_views.home_page, name="home"),
    path("lists/", include("lists.urls")),  ❷
]
```

❶ While we're at it, we use the `import x as y` syntax to alias `views`. This is good practice in your top-level *urls.py*, because it will let us import views from multiple apps if we want—and indeed we will need to later on in the book.

❷ Here's the `include`. Notice that it can take a part of a URL as a prefix, which will be applied to all the included URLs (this is the bit where we reduce duplication, as well as giving our code a better structure).

Back in *lists/urls.py*, we can trim down to only include the latter part of our three URLs, and none of the other stuff from the parent *urls.py*:

lists/urls.py (ch07l058)

```
from django.urls import path
from lists import views

urlpatterns = [
    path("new", views.new_list, name="new_list"),
    path("<int:list_id>/", views.view_list, name="view_list"),
    path("<int:list_id>/add_item", views.add_item, name="add_item"),
]
```

Rerun the unit tests to check that everything worked.

```
Ran 10 tests in 0.040s

OK
```

Can You Believe It?

When I saw this test pass, I couldn't quite believe I did it correctly on the first go. It always pays to be skeptical of your own abilities, so I deliberately changed one of the URLs slightly, just to check if it broke a test. It did. We're covered.

Feel free to try it yourself! Remember to change it back, check that the tests all pass again (including the FTs), and then do a final commit:

```
$ git status
$ git add lists/urls.py
$ git add superlists/urls.py
$ git diff --staged
$ git commit
```

Phew. This was a marathon chapter. But we covered a number of important topics, starting with some thinking about design. We covered rules of thumb like "YAGNI" and "three strikes and refactor". But, most importantly, we saw how to adapt an existing codebase step by step, going from working state to working state, to iterate towards a new design.

I'd say we're pretty close to being able to ship this site, as the very first beta of the superlists website that's going to take over the world. Maybe it needs a little prettification first…let's look at what we need to do to deploy it in the next couple of chapters.

Some More TDD Philosophy

Working state to working state (aka the Testing Goat versus Refactoring Cat)
: Our natural urge is often to dive in and fix everything at once…but if we're not careful, we'll end up like Refactoring Cat, in a situation with loads of changes to our code and nothing working. The Testing Goat encourages us to take one step at a time, and go from working state to working state.

Split work out into small, achievable tasks
: Sometimes this means starting with "boring" work rather than diving straight in with the fun stuff, but you'll have to trust that YOLO-you in the parallel universe is probably having a bad time, having broken everything and struggling to get the app working again.

YAGNI
: You ain't gonna need it! Avoid the temptation to write code that you think *might* be useful, just because it suggests itself at the time. Chances are, you won't use it, or you won't have anticipated your future requirements correctly. See Chapter 24 for one methodology that helps us avoid this trap.

CHAPTER 8

Prettification: Layout and Styling, and What to Test About It

We're starting to think about releasing the first version of our site, but we're a bit embarrassed by how unfinished it looks at the moment. In this chapter, we'll cover some of the basics of styling, including integrating an HTML/CSS framework called Bootstrap. We'll learn how static files work in Django, and what we need to do about testing them.

Testing Layout and Style

Our site is undeniably a bit unattractive at the moment (Figure 8-1).

If you spin up your dev server with `manage.py runserver`, you may run into a database error, something like this: "OperationalError: no such table: lists_list". You need to update your local database to reflect the changes we made in *models.py*. Use `manage.py migrate`. If it gives you any grief about `IntegrityErrors`, just delete the database file.[1]

1 What? Delete the database? Have you taken leave of your senses? Not completely. The local dev database often gets out of sync with its migrations as we go back and forth in our development, and it doesn't have any important data in it, so it's OK to blow it away now and again. We'll be much more careful once we have a "production" database on the server.

We can't be going back to Python's historical reputation for being ugly, so let's do a tiny bit of polishing. Here are a few things we might want:

- A large input field for adding to new and existing lists
- A large, attention-grabbing, centered box to put it in

How do we apply TDD to these things? Most people will tell you that you shouldn't test aesthetics, and they're right. It's a bit like testing a constant, in that tests usually wouldn't add any value.

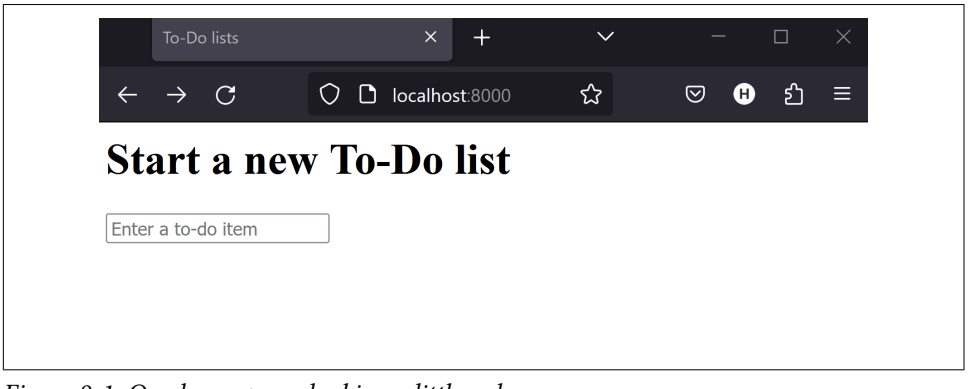


Figure 8-1. Our home page, looking a little ugly…

But we can test the essential *behaviour* of our aesthetics (i.e., that we have any at all). All we want to do is reassure ourselves that things are working. For example, we're going to use Cascading Style Sheets (CSS) for our styling, and they are loaded as static files. Static files can be a bit tricky to configure (especially, as we'll see later, when you move off your own computer and onto a server), so we'll want some kind of simple "smoke test" that the CSS has loaded. We don't have to test fonts and colours and every single pixel, but we can do a quick check that the main input box is aligned the way we want it on each page, and that will give us confidence that the rest of the styling for that page is probably loaded too.

Let's add a new test method inside our functional test (FT):

functional_tests/tests.py (ch08l001)

```
class NewVisitorTest(LiveServerTestCase):
    [...]


    def test_layout_and_styling(self):
        # Edith goes to the home page,
        self.browser.get(self.live_server_url)

        # Her browser window is set to a very specific size
        self.browser.set_window_size(1024, 768)

        # She notices the input box is nicely centered
        inputbox = self.browser.find_element(By.ID, "id_new_item")
        self.assertAlmostEqual(
            inputbox.location["x"] + inputbox.size["width"] / 2,
            512,
            delta=10,
        )
```

A few new things here. We start by setting the window size to a fixed size. We then find the input element, look at its size and location, and do a little maths to check whether it seems to be positioned in the middle of the page. `assertAlmostEqual` helps us to deal with rounding errors and the occasional weirdness due to scrollbars and the like, by letting us specify that we want our arithmetic to work to within 10 pixels, plus or minus.

If we run the FTs, we get:

```
$ python manage.py test functional_tests
[...]
.F.
======================================================================
FAIL: test_layout_and_styling
(functional_tests.tests.NewVisitorTest.test_layout_and_styling)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 119, in
test_layout_and_styling
    self.assertAlmostEqual(
[...]
AssertionError: 102.5 != 512 within 10 delta (409.5 difference)

 ----------------------------------------------------------------------
Ran 3 tests in 9.188s

FAILED (failures=1)
```

That's the expected failure. Still, this kind of FT is easy to get wrong, so let's use a quick-and-dirty "cheat" solution, to check that the FT definitely passes when the input box is centered. We'll delete this code again almost as soon as we've used it to check the FT:

lists/templates/home.html (ch08l002)

```
<form method="POST" action="/lists/new">
  <p style="text-align: center;">
    <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
  </p>
  {% csrf_token %}
</form>
```

That passes, which means the FT works. Let's extend it to make sure that the input box is also center-aligned on the page for a new list:

functional_tests/tests.py (ch08l003)

```
        # She starts a new list and sees the input is nicely
        # centered there too
        inputbox.send_keys("testing")
        inputbox.send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table("1: testing")
        inputbox = self.browser.find_element(By.ID, "id_new_item")
        self.assertAlmostEqual(
            inputbox.location["x"] + inputbox.size["width"] / 2,
            512,
            delta=10,
        )
```

That gives us another test failure:

```
  File "...goat-book/functional_tests/tests.py", line 131, in
test_layout_and_styling
    self.assertAlmostEqual(
AssertionError: 102.5 != 512 within 10 delta (409.5 difference)
```

Let's commit just the FT:

```
$ git add functional_tests/tests.py
$ git commit -m "first steps of FT for layout + styling"
```

Now it feels like we're justified in finding a "proper" solution to improve the styling for our site. We can back out our hacky `text-align: center`:

```
$ git reset --hard
```

`git reset --hard` is the "take off and nuke the site from orbit" Git command, so be careful with it—it blows away all your uncommitted changes. Unlike almost everything else you can do with Git, there's no way of going back after this one.

Prettification: Using a CSS Framework

UI design is hard, and doubly so now that we have to deal with mobile, tablets, and so forth. That's why many programmers, particularly lazy ones like me, turn to CSS frameworks to solve some of those problems for them. There are lots of frameworks out there, but one of the earliest and most popular still, is Bootstrap. Let's use that.

You can find Bootstrap at *getbootstrap.com*.

We'll download it and put it in a new folder called *static* inside the `lists` app:[2]

```
$ wget -O bootstrap.zip https://github.com/twbs/bootstrap/releases/download/\
v5.3.5/bootstrap-5.3.5-dist.zip
$ unzip bootstrap.zip
$ mkdir lists/static
$ mv bootstrap-5.3.5-dist lists/static/bootstrap
$ rm bootstrap.zip
```

Bootstrap comes with a plain, uncustomised installation in the *dist* folder. We're going to use that for now, but you should really never do this for a real site—vanilla Bootstrap is instantly recognisable, and a big signal to anyone in the know that you couldn't be bothered to style your site. Learn how to use Sass and change the font, if nothing else! There is info in Bootstrap's docs, or read an introductory guide.

2 On Windows, you may not have `wget` and `unzip`, but I'm sure you can figure out how to download Bootstrap, unzip it, and put the contents of the *dist* folder into the *lists/static/bootstrap* folder.

Our *lists* folder will end up looking like this:

```
[...]
├── lists
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── migrations
│   │   ├── [...]
│   ├── models.py
│   ├── static
│   │   └── bootstrap
│   │       ├── css
│   │       │   ├── bootstrap-grid.css
│   │       │   ├── bootstrap-grid.css.map
│   │       │   ├── [...]
│   │       │   └── bootstrap.rtl.min.css.map
│   │       └── js
│   │           ├── bootstrap.bundle.js
│   │           ├── bootstrap.bundle.js.map
│   │           ├── [...]
│   │           └── bootstrap.min.js.map
│   ├── templates
│   │   ├── home.html
│   │   └── list.html
│   ├── tests.py
│   ├── urls.py
│   └── views.py
[...]
```

Look at the "Getting started" section of the Bootstrap documentation; you'll see it wants our HTML template to include something like this:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap demo</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
  </body>
</html>
```

We already have two HTML templates. We don't want to be adding a whole load of boilerplate code to each, so now feels like the right time to apply the "Don't repeat yourself" rule, and bring all the common parts together. Thankfully, the Django template language makes that easy using something called template inheritance.

Django Template Inheritance

Let's have a little review of what the differences are between *home.html* and *list.html*:

```
$ diff lists/templates/home.html lists/templates/list.html
<     <h1>Start a new To-Do list</h1>
<     <form method="POST" action="/lists/new">
---
>     <h1>Your To-Do list</h1>
>     <form method="POST" action="/lists/{{ list.id }}/add_item">
[...]
>     <table id="id_list_table">
>       {% for item in list.item_set.all %}
>         <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
>       {% endfor %}
>     </table>
```

They have different header texts, and their forms use different URLs. On top of that, *list.html* has the additional `<table>` element.

Now that we're clear on what's in common and what's not, we can make the two templates inherit from a common "superclass" template. We'll start by making a copy of *list.html*:

```
$ cp lists/templates/list.html lists/templates/base.html
```

We make this into a base template, which just contains the common boilerplate, and mark out the "blocks", places where child templates can customise it:

lists/templates/base.html (ch08l007)

```
<html>
  <head>
    <title>To-Do lists</title>
  </head>

  <body>
    <h1>{% block header_text %}{% endblock %}</h1>

    <form method="POST" action="{% block form_action %}{% endblock %}">
      <input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />
      {% csrf_token %}
    </form>

    {% block table %}
    {% endblock %}
  </body>

</html>
```

Let's see how these blocks are used in practice, by changing *home.html* so that it "inherits" from *base.html*:

lists/templates/home.html (ch08l008)

```
{% extends 'base.html' %}

{% block header_text %}Start a new To-Do list{% endblock %}

{% block form_action %}/lists/new{% endblock %}
```

You can see that lots of the boilerplate HTML disappears, and we just concentrate on the bits we want to customise. We do the same for *list.html*:

lists/templates/list.html (ch08l009)

```
{% extends 'base.html' %}

{% block header_text %}Your To-Do list{% endblock %}

{% block form_action %}/lists/{{ list.id }}/add_item{% endblock %}

{% block table %}
  <table id="id_list_table">
    {% for item in list.item_set.all %}
      <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
    {% endfor %}
  </table>
{% endblock %}
```

That's a refactor of the way our templates work. We rerun the FTs to make sure we haven't broken anything:

```
AssertionError: 102.5 != 512 within 10 delta (409.5 difference)
```

Sure enough, they're still getting to exactly where they were before.

That's worthy of a commit:

```
$ git diff -w
# the -w means ignore whitespace, useful since we've changed some html indenting
$ git status
$ git add lists/templates # leave static, for now
$ git commit -m "refactor templates to use a base template"
```

Integrating Bootstrap

Now it's much easier to integrate the boilerplate code that Bootstrap wants—we won't add the JavaScript yet, just the CSS:

lists/templates/base.html (ch08l010)

```
<!doctype html>
<html lang="en">

  <head>
    <title>To-Do lists</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
[...]
```

Rows and Columns

Finally, let's actually use some of the Bootstrap magic! You'll have to read the documentation yourself, but we should be able to use a combination of the grid system and the `justify-content-center` class to get what we want:

lists/templates/base.html (ch08l011)

```
  <body>
    <div class="container">

      <div class="row justify-content-center">
        <div class="col-lg-6 text-center">
          <h1>{% block header_text %}{% endblock %}</h1>

          <form method="POST" action="{% block form_action %}{% endblock %}">
            <input
              name="item_text"
              id="id_new_item"
              placeholder="Enter a to-do item"
            />
            {% csrf_token %}
          </form>
        </div>
      </div>

      <div class="row justify-content-center">
        <div class="col-lg-6">
          {% block table %}
          {% endblock %}
        </div>
      </div>

    </div>
  </body>
```

(If you've never seen an HTML tag broken up over several lines, that `<input>` may be a little shocking. It is definitely valid, but you don't have to use it if you find it offensive.)

Take the time to browse through the Bootstrap documentation, if you've never seen it before. It's a shopping trolley brimming full of useful tools to use in your site.

Does that work? Whoops—no, we have an error in our unit tests:

```
FAIL: test_renders_input_form
(lists.tests.ListViewTest.test_renders_input_form)
[...]
AssertionError: False is not true : Couldn't find '<input name="item_text"' in
the following response
[...]
```

Ah, it's because our unit tests are currently a little brittle with respect to whitespace changes in our `<input>` tag, which actually don't matter semantically.

Django does provide the `html=True` argument to `assertContains()`, which does help a bit, but it requires exhaustively specifying every attribute of the element we want to check on, like this:

lists/tests.py (ch08l011-1)

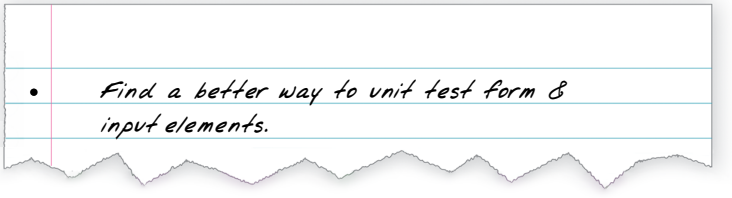
```
class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]

    def test_renders_input_form(self):
        response = self.client.get("/")
        self.assertContains(response, '<form method="POST" action="/lists/new">')
        self.assertContains(
            response,
            '<input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />',
            html=True,
        )
[...]


class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]

    def test_renders_input_form(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")
        self.assertContains(
            response,
            f'<form method="POST" action="/lists/{mylist.id}/add_item">',
        )
        self.assertContains(
            response,
            '<input name="item_text" id="id_new_item" placeholder="Enter a to-do item" />',
            html=True,
        )
```

That's not entirely satisfactory, because all those extra attributes like `id` and `placeholder` aren't really things we want to nail down in unit tests; we'd rather have the freedom to change them in the template without needing to change the tests as well. They're more of a presentation concern than a true part of the contract between backend and frontend.

But it does get the tests to pass:

```
OK
```

So, for now, let's make a note to come back to it:



So, the unit tests are happy. What about the FTs?

```
AssertionError: 102.5 != 512 within 10 delta (409.5 difference)
```

Hmm. No. Why isn't our CSS loading? If you try it manually with `runserver` and look around in DevTools, you'll see the browser 404ing when it tries to fetch *bootstrap.min.css*. If you watch the `runserver` terminal session, you'll also see the 404s there, as in Figure 8-2.

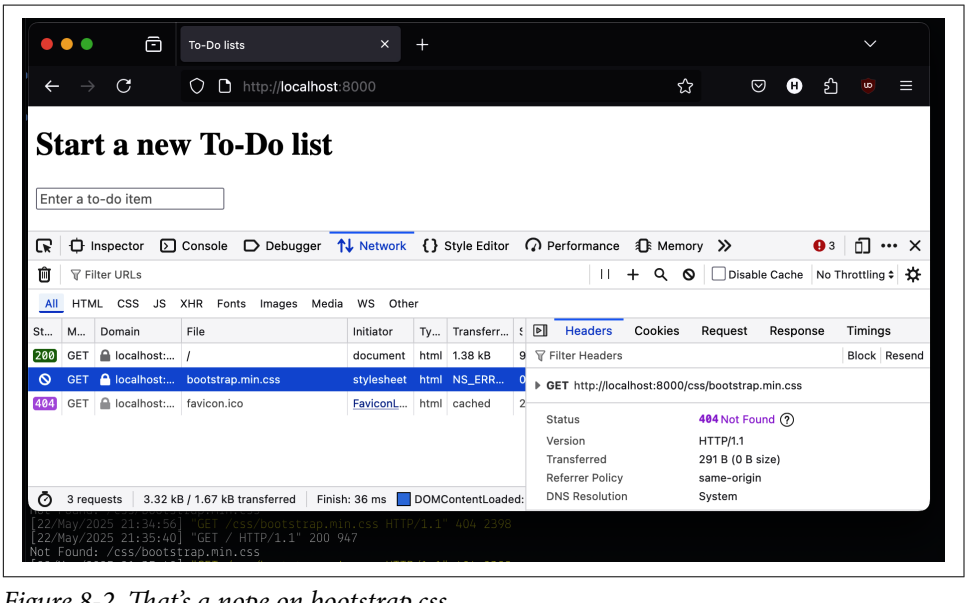


Figure 8-2. That's a nope on bootstrap.css

To figure out what's happening, let's talk a bit about how Django deals with static files.

Static Files in Django

Django, and indeed any web server, needs to know two things to deal with static files:

1. How to tell when a URL request is for a static file, as opposed to for some HTML that's going to be served via a view function
2. Where to find the static file that the user wants

In other words, static files are a mapping from URLs to files on disk.

For item 1, Django lets us define a URL "prefix" to say that any URLs that start with that prefix should be treated as requests for static files. By default, the prefix is */static/*. It's already defined in *settings.py*:

superlists/settings.py

```
[...]

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/5.2/howto/static-files/

STATIC_URL = "static/"
```

The rest of the settings that we will add to this section all have to do with item 2: finding the actual static files on disk.

While we're using the Django development server (`manage.py runserver`), we can rely on Django to magically find static files for us—it'll just look in any subfolder of one of our apps called *static*.

You now see why we put all the Bootstrap static files into *lists/static*. So, why are they not working at the moment? It's because we're not using the `/static/` URL prefix. Have another look at the link to the CSS in *base.html*:

```
    <link href="css/bootstrap.min.css" rel="stylesheet">
```

That `href` is just what happened to be in the Bootstrap docs. To get it to work, we need to change it to:

lists/templates/base.html (ch08l012)

```
    <link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

Now when `runserver` sees the request, it knows that it's for a static file because it begins with `/static/`. It then tries to find a file called *bootstrap/css/bootstrap.min.css*, looking in each of our app folders for subfolders called *static*, and it should find it at *lists/static/bootstrap/css/bootstrap.min.css*.

So if you take a look manually, you should see it works, as in Figure 8-3.

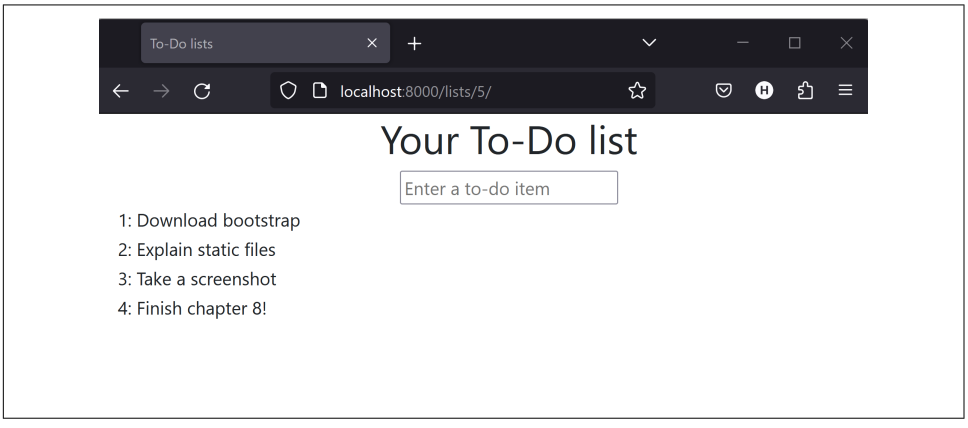


Figure 8-3. Our site starts to look a little better…

Switching to StaticLiveServerTestCase

If you run the FT though, annoyingly, it still won't pass:

```
AssertionError: 102.5 != 512 within 10 delta (409.5 difference)
```

That's because, although `runserver` automagically finds static files, `LiveServerTestCase` doesn't. Never fear, though: the Django developers have made an even more magical test class called `StaticLiveServerTestCase` (see the docs).

Let's switch to that:

functional_tests/tests.py (ch08l013)

```
@@ -1,14 +1,14 @@
-from django.test import LiveServerTestCase
+from django.contrib.staticfiles.testing import StaticLiveServerTestCase
 from selenium import webdriver
 from selenium.common.exceptions import WebDriverException
 from selenium.webdriver.common.keys import Keys
 import time

 MAX_WAIT = 10


-class NewVisitorTest(LiveServerTestCase):
+class NewVisitorTest(StaticLiveServerTestCase):

     def setUp(self):
```

And now it will find the new CSS, which will get our test to pass:

```
$ python manage.py test functional_tests
Creating test database for alias 'default'...
...
 ----------------------------------------------------------------------
Ran 3 tests in 9.764s
```

Hooray!

Using Bootstrap Components to Improve the Look of the Site

Let's see if we can do even better, using some of the other tools in Bootstrap's panoply.

Jumbotron!

The first version of Bootstrap used to ship with a class called `jumbotron` for things that are meant to be particularly prominent on the page. It doesn't exist anymore, but old-timers like me still pine for it, so they have a specific page in the docs that tells you how to re-create it.

Essentially, we massively embiggen the main page header and the input form, putting it into a grey box with nice rounded corners:

lists/templates/base.html (ch08l014)

```
<body>
  <div class="container">

    <div class="row justify-content-center p-5 bg-body-tertiary rounded-3">
      <div class="col-lg-6 text-center">
        <h1 class="display-1 mb-4">{% block header_text %}{% endblock %}</h1>
        [...]
```

That ends up looking something like Figure 8-4.

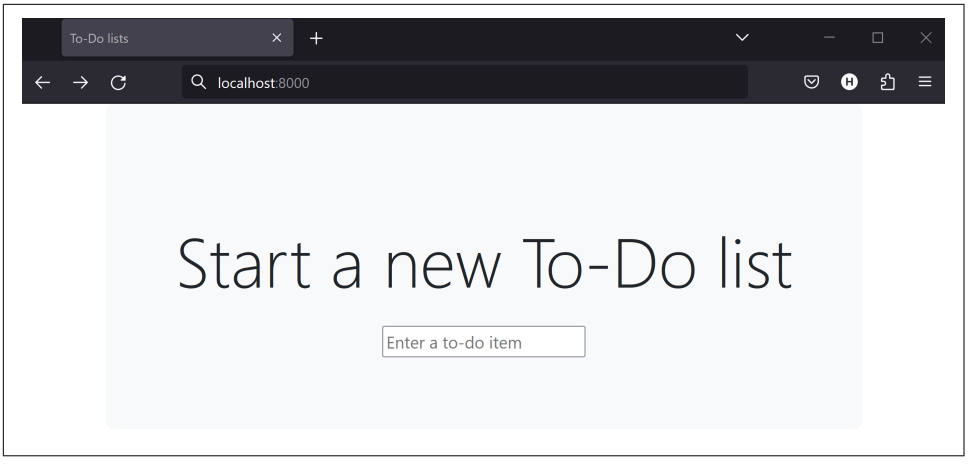


Figure 8-4. A big grey box at the top of the page

When hacking about with design and layout, it's best to have a window open that we can refresh frequently. Use `python manage.py runserver` to spin up the dev server, and then browse to *http://localhost:8000* to see your work as we go.

Large Inputs

The `jumbotron` is a good start, but now the input box has tiny text compared to everything else. Thankfully, Bootstrap's form control classes offer an option to set an input to "large":

lists/templates/base.html (ch08l015)

```
<input
  class="form-control form-control-lg"
  name="item_text"
  id="id_new_item"
  placeholder="Enter a to-do item"
/>
```

Table Styling

The table text also looks too small compared to the rest of the page now. Adding the Bootstrap `table` class improves things, over in *list.html*:

lists/templates/list.html (ch08l016)

```
<table class="table" id="id_list_table">
```

Optional: Dark Mode

In contrast to my greybeard nostalgia for `jumbotron`, here's something relatively new to Bootstrap: dark mode!

lists/templates/base.html (ch08l017)

```
<!doctype html>
<html lang="en" data-bs-theme="dark">
```

Take a look at Figure 8-5. I think that looks great!

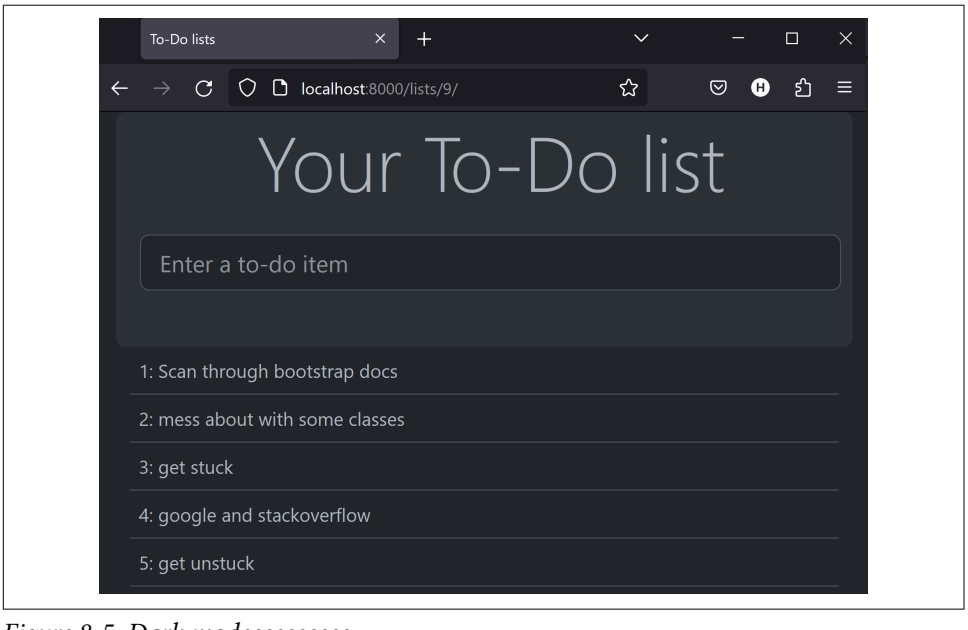


Figure 8-5. Dark modeeeeeeeeee

But it's very much a matter of personal preference, and my editor will have kittens if I make all the rest of my screenshots use so much ink, so I'm going to revert it for now. You're free to keep dark mode on if you like!

A Semi-Decent Page

Getting it into shape took me a few goes, but I'm reasonably happy with it now (Figure 8-6).

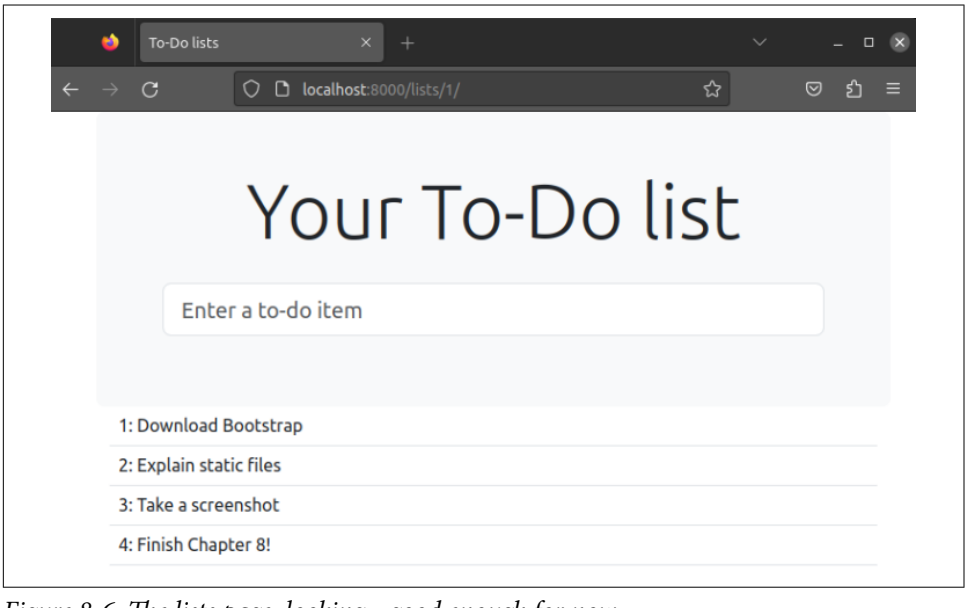


Figure 8-6. The lists page, looking…good enough for now

If you want to go further with customising Bootstrap, you need to get into compiling Sass. I've said it already, but I *definitely* recommend taking the time to do that someday. Sass/SCSS is a great improvement on plain old CSS, and a useful tool even if you don't use Bootstrap.

A last run of the FTs, to see if everything still works OK:

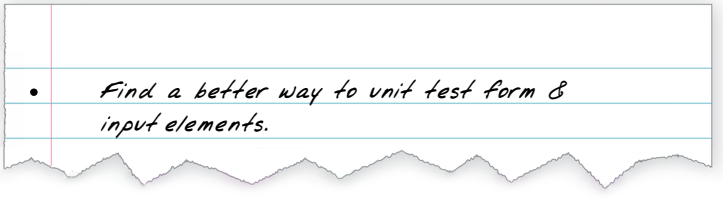
```
$ python manage.py test functional_tests
[...]
...
 ---------------------------------------------------------------------
Ran 3 tests in 10.084s

OK
```

That's it! Definitely time for a commit:

```
$ git status # changes tests.py, base.html, list.html, settings.py,
             # and untracked lists/static
$ git add .
$ git status # will now show all the bootstrap additions
$ git commit -m "Use Bootstrap to improve layout"
```

Parsing HTML for Less Brittle Tests of Key HTML Content

Oh whoops, we nearly forgot our scratchpad:



When working on layout and styling, you expect to spend most of your time in the browser, in a cycle of tweaking your HTML and refreshing to see the effects, with occasional runs of your layout FT, if you have one.

You wouldn't expect to test-drive design with unit tests. And sure enough, we haven't run them in a while. Because if we had done, we'd have noticed that they're failing:

```
FAIL: test_renders_input_form
(lists.tests.HomePageTest.test_renders_input_form)
[...]
AssertionError: False is not true : Couldn't find '<input name="item_text"
id="id_new_item" placeholder="Enter a to-do item" />' in the following response
b'<!doctype html>\n<html lang="en">\n\n  <head>\n    <title>To-Do
[...]
<input\n              class="form-control form-control-lg"\n
name="item_text"\n              id="id_new_item"\n
placeholder="Enter a to-do item"\n            />\n            <input
[...]
FAIL: test_renders_input_form
(lists.tests.ListViewTest.test_renders_input_form)
[...]
```

It's also annoyingly hard to see from the tests output, but it happened when we introduced the `class=form-control form-control-lg`.

We really don't want this sort of thing breaking our unit tests. Using string matching, even whitespace-aware string matching, is just the wrong tool for the job.[3] Let's switch to using a proper HTML parser, the venerable lxml.

3 As famously explained in a classic Stack Overflow post.

```
$ pip install 'lxml[cssselect]'
Collecting lxml[cssselect]
  [...]
Collecting cssselect>=0.7 (from lxml[cssselect])
  [...]
Installing collected packages: lxml, cssselect
Successfully installed [...]
```

(We need the `cssselect` add-on for the nice CSS selectors.)

And here's how we use it to write a more focused version of our test that only cares about the two HTML attributes that actually matter to the integration of frontend and backend:

1. The `<form>` tag's `method` and `action`
2. The `<input>` tag's `name`

lists/tests.py (ch08l019)

```
import lxml.html
[...]


class HomePageTest(TestCase):
    def test_uses_home_template(self):
        [...]

    def test_renders_input_form(self):
        response = self.client.get("/")
        parsed = lxml.html.fromstring(response.content)  ❶
        [form] = parsed.cssselect("form[method=POST]")  ❷❸
        self.assertEqual(form.get("action"), "/lists/new")
        [input] = form.cssselect("input[name=item_text]")  ❹
```

❶ Here's where we parse the HTML into a structured object to represent the DOM (document object model).

❷ Here's where we use a CSS selector to find our form, implicitly also checking that it has `method="POST"`. The `cssselect()` method returns a list of matching elements.

❸ The `[form] =` is worth a mention. What we're using here is a special assignment syntax called "unpacking", where the lefthand side is a list of variable names and the righthand side is a list of values. It's a bit like saying `form = parsed.cssselect("form[method=POST]")[0]`, but a bit nicer to read, and a bit more strict

too. By only putting one element on the left, we're effectively asserting that there is exactly one element on the right; if there isn't, we'll get an error.[4]

❹ We use the same kind of assignment to assert that the form contains exactly one input element with the name `item_text`.

Here's the same thing in `ListViewTest`:

lists/tests.py (ch08l020)

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]

    def test_renders_input_form(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")
        parsed = lxml.html.fromstring(response.content)
        [form] = parsed.cssselect("form[method=POST]")
        self.assertEqual(form.get("action"), f"/lists/{mylist.id}/add_item")
        [input] = form.cssselect("input[name=item_text]")
```

That works!

```
Ran 10 tests in 0.017s

OK
```

And as always, for any test you've only ever seen green, it's nice to introduce a deliberate failure:

lists/templates/base.html (ch08l021)

```
@@ -18,7 +18,7 @@
           <form method="POST" action="{% block form_action %}{% endblock %}">
             <input
               class="form-control form-control-lg"
-              name="item_text"
+              name="geoff"
               id="id_new_item"
               placeholder="Enter a to-do item"
             />
```

4 Read more about tuple unpacking and multiple assignment on Trey Hunner's excellent blog.

And let's see the error message:

```
    [input] = form.cssselect("input[name=item_text]")
    ^^^^^^^
ValueError: not enough values to unpack (expected 1, got 0)
```

Hmm you know what? I'm actually not happy with that. The `[input] =` syntax is probably another example of me being too clever for my own good.

Let's try something else that will give us a clearer message about what *is* on the page and what isn't:

lists/tests.py (ch08l022)

```
        inputs = form.cssselect("input")  ❶
        self.assertIn("item_text", [input.get("name") for input in inputs])  ❷
```

❶ We'll get a list of all the inputs in the form.

❷ And then we'll assert that at least one of them has the right `name=`.

That gives us a more self-explanatory message:

```
    self.assertIn("item_text", [input.get("name") for input in inputs])
    ~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'item_text' not found in ['geoff', 'csrfmiddlewaretoken']
```

Now I feel good about changing our HTML back:

lists/templates/base.html (ch08l023)

```
@@ -18,7 +18,7 @@
           <form method="POST" action="{% block form_action %}{% endblock %}">
             <input
               class="form-control form-control-lg"
-              name="geoff"
+              name="item_text"
               id="id_new_item"
               placeholder="Enter a to-do item"
             />
```

Much better!

```
$ git diff # tests.py
$ git commit -am "use lxml for more specific unit test asserts on html content"
```

What We Glossed Over: collectstatic and Other Static Directories

We saw earlier that the Django dev server will magically find all your static files inside app folders, and serve them for you. That's fine during development, but when you're running on a real web server, you don't want Django serving your static content—using Python to serve raw files is slow and inefficient, and a web server like Apache or nginx can do this all for you.

For these reasons, you want to be able to gather all your static files from inside their various app folders and copy them into a single location, ready for deployment. This is what the `collectstatic` command is for.

The destination, the place where the collected static files go, needs to be defined in *settings.py* as `STATIC_ROOT`. In the next chapter, we'll be doing some deployment, so let's actually experiment with that now. A common and straightforward place to put it is in a folder called "static" in the root of our repo:

```
.
├── db.sqlite3
├── functional_tests/
├── lists/
├── manage.py
├── static/
└── superlists/
```

Here's a neat way of specifying that folder, making it relative to the location of the project base directory:

superlists/settings.py (ch08l024)

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/5.2/howto/static-files/

STATIC_URL = "static/"
STATIC_ROOT = BASE_DIR / "static"
```

Take a look at the top of the settings file, and you'll see how that `BASE_DIR` variable is helpfully defined for us, using `pathlib.Path` and `__file__` (both really nice Python built-ins).[5]

5 Notice in the `Pathlib` wrangling of `__file__` that the `.resolve()` happens before anything else. Always follow this pattern when working with `__file__`, otherwise you can see unpredictable behaviours depending on how the file is imported. Thanks to Green Nathan for that tip!

Anyway, let's try running `collectstatic`:

```
$ python manage.py collectstatic

171 static files copied to '...goat-book/static'.
```

And if we look in *./static*, we'll find all our CSS files:

```
$ tree static/
static/
├── admin
│   ├── css
│   │   ├── autocomplete.css
│   │   ├── [...]
[...]
│               └── xregexp.min.js
└── bootstrap
    ├── css
    │   ├── bootstrap-grid.css
    │   ├── [...]
    │   └── bootstrap.rtl.min.css.map
    └── js
        ├── bootstrap.bundle.js
        ├── [...]
        └── bootstrap.min.js.map

17 directories, 171 files
```

`collectstatic` has also picked up all the CSS for the admin site. The admin site is one of Django's powerful features, but we don't need it for our simple site, so let's disable it for now:

superlists/settings.py (ch08l025)

```
INSTALLED_APPS = [
    # "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "lists",
]
```

And we try again:

```
$ rm -rf static/
$ python manage.py collectstatic

44 static files copied to '...goat-book/static'.
```

Much better.

Now we know how to collect all the static files into a single folder, where it's easy for a web server to find them. We'll find out all about that, including how to test it, in the next chapter!

For now, let's save our changes to *settings.py*. We'll also add the top-level static folder to our `gitignore`, because it will only contain copies of files we actually keep in individual apps' static folders:

```
$ git diff # should show changes in settings.py
$ echo /static >> .gitignore
$ git commit -am "set STATIC_ROOT in settings and disable admin"
```

A Few Things That Didn't Make It

Inevitably this was only a whirlwind tour of styling and CSS, and there were several topics that I'd considered covering that didn't make it. Here are a few candidates for further study:

- The `{% static %}` template tag, for more DRY and fewer hardcoded URLs
- Client-side packaging tools, like `npm` and `bower`
- Customising Bootstrap with Sass

Recap: On Testing Design and Layout

The tl;dr is: you shouldn't write tests for design and layout per se. It's too much like testing a constant, and the tests you write are often brittle.

With that said, the *implementation* of design and layout involves something quite tricky: CSS and static files. As a result, it is valuable to have some kind of minimal "smoke test" that checks that your static files and CSS are working. As we'll see in the next chapter, it can help pick up problems when you deploy your code to production.

Similarly, if a particular piece of styling required a lot of client-side JavaScript code to get it to work (dynamic resizing is one I've spent a bit of time on), you'll definitely want some tests for that (see Chapter 17).

Try to write the minimal tests that will give you the confidence that your design and layout is working, without testing *what* it actually is. That includes unit tests! Avoid asserting on the cosmetic aspects of your HTML in your unit tests.

Aim to leave yourself in a position where you can freely make changes to the design and layout, without having to go back and adjust tests all the time.

PART II
Going to Production

> Is all fun and game until you are need of put it in production.
>
> —DevOps Borat

It's time to deploy the first version of our site and make it public. They say that if you wait until you feel *ready* to ship, then you've waited too long.

Is our site usable? Is it better than nothing? Can we make lists on it? Yes, yes, yes.

No, you can't log in yet. No, you can't mark tasks as completed. But do we really need any of that stuff? Not really—and you can never be sure what your users are *actually* going to do with your site once they get their hands on it. We think our users want to use the site for to-do lists, but maybe they actually want to use it to make "top 10 best fly-fishing spots" lists, for which you don't *need* any kind of "mark completed" function. We won't know until we put it out there.

Over the next couple of chapters we're going to go through and actually deploy our site to a real, live web server.

You might be tempted to skip this bit—there's lots of daunting stuff in it, and maybe you think this isn't what you signed up for. But I *strongly* urge you to give it a go. This is one of the sections of the book I'm most pleased with, and it's one that people often write to me about saying they were really glad they stuck through it.

If you've never done a server deployment before, it will demystify a whole world for you, and there's nothing like the feeling of seeing your site live on the actual internet. Give it a buzzword name like "DevOps" if that's what it takes to convince you it's worth it.

The Danger Areas of Deployment

Deploying a site to a live web server can be a tricky topic. Oft heard is the forlorn cry, "but it works on my machine!"

Some of the danger areas of deployment include:

Networking
: Once we're off our own machine, networking issues come in: making sure that DNS is routing our domain to the correct IP address for our server, making sure our server is configured to listen to traffic coming in from the world, making sure it's using the right ports, and making sure any firewalls in the way are configured to let traffic through.

Dependencies
: We need to make sure that the packages our software relies on (Python, Django, and so on) are installed on the server and have the correct versions.

The database
: There can be permissions and path issues, and we need to be careful about preserving data between deploys.

Static files (CSS, JavaScript, images, etc.)
: Web servers usually need special configuration for serving these.

Security and configuration
: Once we're on the public internet, we need to worry more about security. Various settings that are really useful for local development (like the Django debug page) become dangerous in production (because they expose our source code in tracebacks).

Reproducibility and divergence between local dev and prod
: All of the above add up to differences between your local development environment and the way code runs in production. We want to be able to reproduce the way things work on our machine, as closely as possible, in production (and vice versa) to give us as much confidence as possible that "it works on my machine" means "it's going to work in production".

One way to approach the problem is to get a server and start manually configuring and installing everything, hacking about until it works, and maybe think about automating things later.[1]

1 This was, more or less, the approach I took in earlier editions of the book. With a fair bit of testing thrown in, of course.

But if there's one thing we've learned in the world of Agile/Lean software development, it's that taking smaller steps usually pays off.

How can we take smaller, safer steps towards a production deployment? Can we *simulate* the process of moving to a server so that we can iron out all the bugs before we actually take the plunge? Can we then make small changes one at a time, solving problems one by one, rather than having to bite off everything in one mouthful? Can we use our existing test suite to make sure things work on the server, as well as locally?

Absolutely we can. And if you've looked at the table of contents, I'm sure you're already guessing that Docker is going to be part of the answer.

An Overview of Our Deployment Procedure

Over the next three chapters, I'm going to go through a deployment procedure. It isn't meant to be the *perfect* deployment procedure, so please don't take it as being best practice or a recommendation—it's meant to be an illustration, to show the kinds of issues involved in putting code into production, and where testing fits in.

Chapter 9

- Adapt our functional tests (FTs) so they can run against a container.
- Build a minimal Dockerfile with everything we need to run our site.
- Learn how to build and run a container on our machine.
- Get a first cut of our code up and running inside Docker, with passing tests.

Chapter 10

- Gradually, incrementally change the container configuration to make it production-ready.
- Regularly rerun the FTs to check we didn't break anything.
- Address issues to do with the database, static files, secrets, and so on.

Chapter 11

- Set up a "staging" server,[1] using the same infrastructure that we plan to use for production.
- Set up a real domain name and point it at this server.
- Install Ansible and flush out any networking issues.

1 Some people prefer the term pre-prod or test environment. It's all the same idea.

- Gradually build up an Ansible playbook to deploy our containers on a real server.
- Again, use our FTs to check for any problems.
- Learn how to SSH (Secure Shell) into the server to debug things, locate logs, and find other useful information.
- Confidently deploy to production once we have a working deployment script for staging.

TDD and Docker Versus the Danger Areas of Deployment

Hopefully you can start to see how the combination of TDD, Docker, staging, and automation are going to help minimise the risk of the various "danger areas":

Containers as mini servers
: Containers will act as mini servers letting us flush out issues with dependencies, static files, and so on. A key advantage is that they'll give us a way of getting faster feedback cycles; because we can spin them up locally almost instaneously, we can very quicly see the effect of any changes.

Packaging Python and system dependencies
: Our containers will package up both our Python and system dependencies, including a production-ready web server and static files system, as well as many production settings and configuration differences. This minimises the difference between what we can test locally, and what we will have on our servers. As we'll see, it will give us a reliable way to reproduce bugs we see in production, on our local machine.

Fully automated FTs
: Our FTs mean that we'll have a fully automated way of checking that everything works.

Running FTs on staging server
: Later, when we deploy our containers to a staging server, we can run the FTs against that too. It'll be slightly slower and might involve some fiddly compromises, but it'll give us one more layer of reassurance.

Automating build and deployment
: Finally, by fully automating container creation and deployment, and by testing the end results of both these things, we maximise reproducibility, thus minimising the risk of deployment to production.

Oh, but there's lots of fun stuff coming up! Just you wait!

CHAPTER 9

Containerization aka Docker

> Little boxes, all the same
>
> —Malvina Reynolds

In this chapter, we'll start by adapting our FTs so that they can run against a container. And then we'll set about containerising our app, and getting those tests passing our code running inside Docker:

- We'll build a minimal Dockerfile with everything we need to run our site.
- We'll learn how to build and run a container on our machine.
- We'll make a few changes to our source code layout, like using a *src* folder.
- We'll start flushing out a few issues around networking and the database.

Docker, Containers, and Virtualization

Docker is a commercial product that wraps several free and open source technologies from the world of Linux, sometimes referred to as "containerization".

Feel free to skip this section if you already know all about Docker.

You may have already heard of the idea of "virtualization", which enables a single physical computer to pretend to be several machines. Pioneered by IBM (amongst others) on mainframes in the 1960s, it rose to mainstream adoption in the '90s, where it was sold as a way to optimise resource usage in datacentres. AWS, for example, an

offshoot of Amazon, was using virtualization already, and realised it could sell some spare capacity on its servers to customers outside the business.

So, when you come to deploy your code to a real server in a datacentre, it will be using virtualization. And, actually, you can use virtualization on your own machine, with software like VirtualBox or KVM. You can run Windows "inside" a Mac or Linux laptop, for example.

But it can be fiddly to set up! And nowadays, thanks to containerization, we can do better because containerization is a kind of even-more-virtual virtualization.

Conceptually, "regular" virtualization works at the hardware level: it gives you multiple virtual machines (VMs) that pretend to be different physical computers, on a single real machine. So you can run multiple operating systems using separate VMs on the same physical box, as in Figure 9-1.

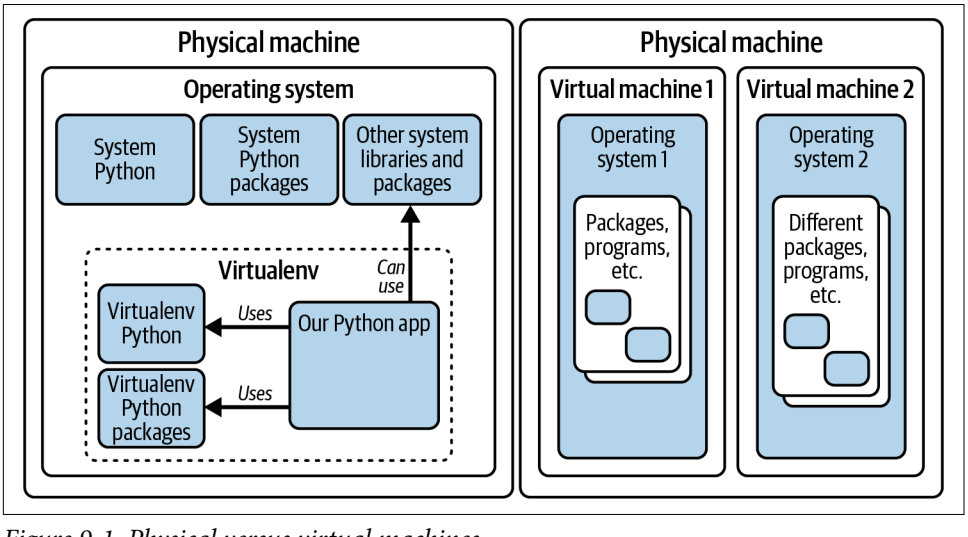


Figure 9-1. Physical versus virtual machines

Containerization works at the operating system (OS) level: it gives you multiple virtual operating systems that all run on a single real OS.[1]

1 It's more accurate to say that containers share the same kernel as the host OS. An operating system is made up of a kernel, and a bunch of utility programs that run on top of it. The kernel is the core of the OS; it's the program that runs all the other programs. Whenever your program needs to interact with the outside world, read a file, or talk to the internet, or start another program, it actually asks the kernel to do it. Starting about 15 years ago, the Linux kernel grew the ability to show different filesystems to different programs, as well as isolate them into different network and process namespaces; these are the capabilities that underpin Docker and containerization.

Containers let us pack the source code and the system dependencies (like Python or system libraries) together, and our programs run inside separate virtual systems, using a single real host OS and kernel.[2] See Figure 9-2 for an illustration.

The upshot of this is that containers are much "cheaper". You can start one up in milliseconds, and you can run hundreds on the same machine.

If you're new to all this, I know it's a lot to wrap your head around! It takes a while to build a good mental model of what's happening. Have a look at Docker's resources on containers for more explanation. Hopefully, following along with these chapters and seeing them working in practice will help you to better understand the theory.

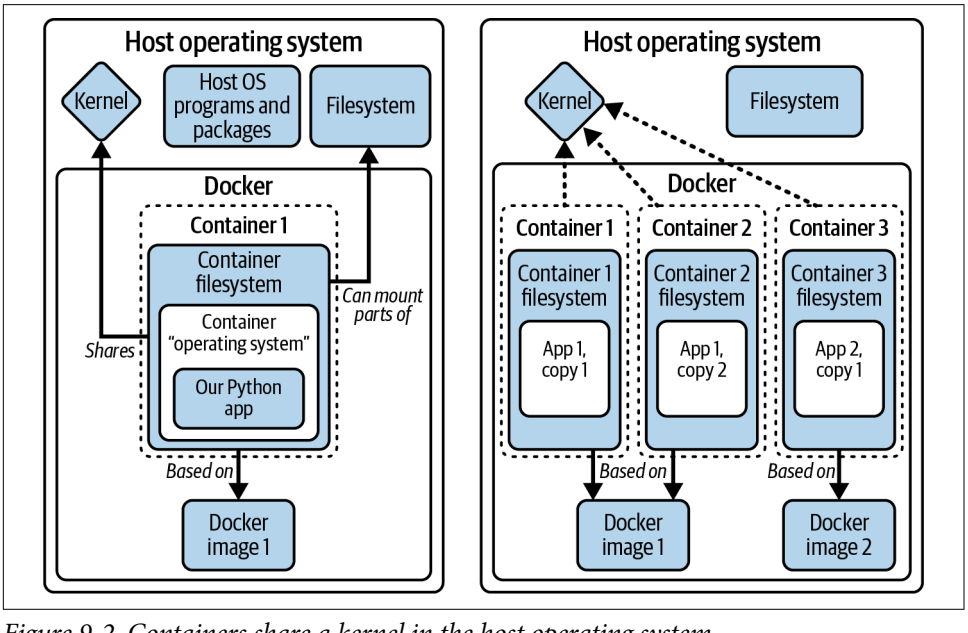


Figure 9-2. Containers share a kernel in the host operating system

2 Because containers all share the same kernel, while virtualization can let you run Windows and Linux on the same machine, containers on Linux hosts all run Linux, and ones on Windows hosts all run Windows. If you're running Linux containers on a Mac or a PC, it's because you're actually running them on a Linux VM under the hood.

Why Not Just Use a Virtualenv?

You might be thinking that this sounds a lot like a virtualenv—and you'd be right! Virtualenvs already let us run different versions of Python, with different Python packages, on the same machine.

What Docker containers give us over and above virtualenvs, is the ability to have different *system* dependencies too; things you can't `pip install`, in other words. In the Python world, this could be C libraries, like `libpq` for PostgreSQL, or `libxml2` for parsing XML. But you could also run totally different programming languages in different containers, or even different Linux distributions. So, server administrators or platform people like them because it's one system for running any kind of software, and they don't need to understand the intricacies of any particular language's packaging systems.

Docker and Your CV

That's all well and good for the *theoretical* justification, but let's get to the *real* reason for using this technology, which, as always, is: "it's fashionable so it's going to look good on my CV".

For the purposes of this book, that's not such a bad justification really!

Yes, it's going to be a nice way to have a "pretend" deployment on our own machine, before we try the real one—but also, containers are so popular nowadays, that it's very likely that you're going to encounter them at work (if you haven't already). For many working developers, a container image is the final artifact of their work; it's what they "deliver", and often the rest of the deployment process is something they rarely have to think about.

In any case, without further ado, let's get into it!

As Always, Start with a Test

Let's adapt our functional tests (FTs) so that they can run against a standalone server, instead of the one that `LiveServerTestCase` creates for us.

Do you remember I said that `LiveServerTestCase` had certain limitations? Well, one is that it always assumes you want to use its own test server, which it makes available at `self.live_server_url`. I still want to be able to do that *sometimes*, but I also want to be able to selectively tell it not to bother, and to use a real server instead.

We'll do it by checking for an environment variable called `TEST_SERVER`:

functional_tests/tests.py (ch09l001)

```
import os
[...]

class NewVisitorTest(StaticLiveServerTestCase):
    def setUp(self):
        self.browser = webdriver.Firefox()
        if test_server := os.environ.get("TEST_SERVER"):  ❶❷
            self.live_server_url = "http://" + test_server  ❸
```

❶ Here's where we check for the env var.

❷ If you haven't seen this before, the `:=` is known as the "walrus operator" (more formally, it's the operator for an "assignment expression"), which was a controversial new feature from Python 3.8[3] and it's not often useful, but it is quite neat for cases like this, where you have a variable and want to do a conditional on it straight away. See this article for more explanation.

❸ Here's the hack: we replace `self.live_server_url` with the address of our "real" server.

A clarification: when we say we run tests *against* our Docker container, or *against* our staging server, that doesn't mean we run the tests *from* Docker or *from* our staging server. We still run the tests from our own laptop, but they target the place that's running our code.

We test that said hack hasn't broken anything by running the FTs "normally":

```
$ python manage.py test functional_tests
[...]
Ran 3 tests in 8.544s

OK
```

And now we can try them against our Docker server URL—which, once we've done the right Docker magic, will be at *http://localhost:8888*.

3 The feature was a favourite of Guido van Rossum's, but the discussion around it was so toxic that Guido stepped down from his role as Python's BDFL, or "Benevolent Dictator for Life".

I'm deliberately choosing a different port to run Dockerised Django on (8888) from the default port that a local `manage.py runserver` would choose (8080). This is to avoid getting in the situation where I (or the tests) *think* we're looking at Docker, when we're actually looking at a local `runserver` that I've left running in some terminal somewhere.

Ports

Ports are what let you have multiple connections open at the same time on a single machine; the reason you can load two different websites at the same time, for example.

Each network adapter has a range of ports, numbered from 0 to 65535. In a client/server connection, the client knows the port of the server, and the client OS chooses a random local port for its side of the connection.

When a server is "listening" on a port, no other service can bind to that port at the same time. That's why you can't run `manage.py runserver` in two different terminals at the same time, because both want to use port `8080` by default.

We'll use the `--failfast` option to exit as soon as a single test fails:

```
$ TEST_SERVER=localhost:8888 ./manage.py test functional_tests --failfast
[...]
E
======================================================================
ERROR: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/tests.py", line 38, in
test_can_start_a_todo_list
    self.browser.get(self.live_server_url)
[...]

selenium.common.exceptions.WebDriverException: Message: Reached error page: abo
ut:neterror?e=connectionFailure&u=http%3A//localhost%3A8888/[...]


Ran 1 tests in 5.518s

FAILED (errors=1)
```

If, on Windows, you see an error saying something like "TEST_SERVER is not recognized as a command", it's probably because you're not using Git Bash. Take another look at the "Prerequisites and Assumptions" section.

You can see that our tests are failing, as expected, because we're not running Docker yet. Selenium reports that Firefox is seeing an error and "cannot establish connection to the server", and you can see *localhost:8888* in there too.

The FT seems to be testing the right things, so let's commit:

```
$ git diff # should show changes to functional_tests.py
$ git commit -am "Hack FT runner to be able to test docker"
```

Don't use `export` to set the `TEST_SERVER` environment variable; otherwise, all your subsequent test runs in that terminal will be against staging, and that can be very confusing if you're not expecting it. Setting it explicitly inline each time you run the FTs is best.

Making a src Folder

When preparing a codebase for deployment, it's often convenient to separate out the actual source code of our production app from the rest of the files that you need in the project. A folder called *src* is a common convention.

Currently, all our code is source code really, so we move everything into *src* (we'll be seeing some new files appearing outside *src* shortly):[4]

```
$ mkdir src
$ git mv functional_tests lists superlists manage.py src
$ git commit -m "Move all our code into a src folder"
```

Installing Docker

The Docker documentation is pretty good, and you'll find detailed installation instructions for Windows, Mac, and Linux.

Choose WSL (Windows Subsystem for Linux) as your backend on Windows, as we'll need it in the next chapter. You can find installation instructions on the Microsoft website. This doesn't mean you have to switch your development environment to being "inside" WSL; Docker just uses WSL as a virtualization engine in the background. You should be able to run all the `docker` CLI commands from the same Git Bash console you've been using so far.

4 A common thing to find outside of the *src* folder is a folder called *tests*. We won't be doing that while we're relying on the standard Django test framework, but it can be a good thing to do if you're using pytest, for example.

Docker Alternatives: Podman, nerdctl, etc.

Impartiality commands me to also mention Podman and nerdctl, both like-for-like replacements for Docker.

They are both pretty much exactly the same as Docker, arguably with a few advantages even.[5]

I actually tried Podman out on early drafts of this chapter (on Linux) and it worked perfectly well. But they are both a little less well established and documented; the Windows installation instructions are a little more DIY, for example. So in the end, although I'm always a fan of a plucky noncommercial upstart, I decided to stick with Docker for now. After all, the core of it is still open source, to its credit! But you could definitely check out one of the alternatives if you feel like it.

You can follow along all the instructions in the book by just substituting the `docker` binary for `podman` or `nerdctl` in all the CLI instructions:

```
$ docker run busybox echo hello
# becomes
$ podman run busybox echo hello
# or
$ nerdctl run busybox echo hello
# similarly with podman build, nerdtcl build, podman ps, etc.
```

Colima: An Alternative Docker Runtime for macOS

If you're on macOS, you might find the Docker Dekstop licensing terms don't work for you. In that case, you can try Colima, which is a "container runtime", essentially the backend for Docker. You still use the Docker CLI tools, but Colima provides the server to run the containers:

```
$ docker run busybox echo hello
docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock.
Is the docker daemon running?.
See docker run --help.
$ colima start
INFO[0001] starting colima
INFO[0001] runtime: docker
INFO[0001] starting ...                                  context=vm
INFO[0014] provisioning ...                              context=docker
INFO[0016] starting ...                                  context=docker
INFO[0017] done
$ docker run busybox echo hello
hello
```

5 Docker uses a central "daemon" to manage containers, which Podman and nerdctl don't.

I used Colima for most of the writing of this book, and it worked fine for me. The only thing I needed to do was set the `DOCKER_HOST` environment variable, and that only came up in Chapter 12:

```
$ *export DOCKER_HOST=unix:///$HOME/.colima/default/docker.sock
```

On macOS, you can use Colima as a backend for nerdctl. Podman ships with its own runtime, for both Mac and Windows (there is no need for a runtime on Linux).

At the time of writing, Apple had just announced its own container runner, *container*, but it was in beta and I didn't have time to try it out.

Test your installation by running:

```
$ docker run busybox echo hello world
Unable to find image 'busybox:latest' locally
[...]
latest: Pulling from library/busybox
[...]: Pull complete
Digest: sha256:[...]
Status: Downloaded newer image for busybox:latest
hello world
```

What's happened there is that Docker has:

- Searched for a local copy of the "busybox" image and not found it
- Downloaded the image from Docker Hub
- Created a container based on that image
- Started up that container, telling it to run `echo hello world`
- And we can see it worked!

Cool! We'll find out more about all of these steps as the chapter progresses.

On macOS, if you get errors saying `command not found: docker`, obviously the first thing you should do is Google for "macOS command not found Docker", but at least one reader has reported that the solution was Docker Desktop > Settings > Advanced > Change from User to System.

Building a Docker Image and Running a Docker Container

Docker has the concepts of *images* as well as containers. An image is essentially a pre-prepared root filesystem, including the OS, dependencies, and any code you want to run.

Once you have an image, you can run one or more containers that use the same image. It's a bit like saying, once you've installed your OS and software, you can start up your computer and run that software any number of times, without needing to change anything else.

Another way of thinking about it is: images are like classes, and containers are like instances.

A First Cut of a Dockerfile

Think of a Dockerfile as instructions for setting up a brand new computer that we're going to use to run our Django server on. What do we need to do? Something like this, right?

1. Install an operating system.
2. Make sure it has Python on it.
3. Get our source code onto it.
4. Run `python manage.py runserver`.

We create a new file called *Dockerfile* in the base folder of our repo, next to the *src/* directory we made earlier:

Dockerfile (ch09l003)

```
FROM python:3.14-slim  ❶

COPY src /src  ❷

WORKDIR /src  ❸

CMD ["python", "manage.py", "runserver"]  ❹
```

❶ The `FROM` line is usually the first thing in a Dockerfile, and it says which *base image* we are starting from. Docker images are built from other Docker images! It's not quite turtles all the way down, but almost. So this is the equivalent of choosing a base OS, but images can actually have lots of software preinstalled too. You can browse various base images on Docker Hub. We're using one that's published by the Python Software Foundation, called "slim" because it's as small as possible. It's based on a popular version of Linux called Debian, and of course it comes with Python already installed on it.

❷ The `COPY` instruction (the uppercase words are called "instructions") lets you copy files from your own computer into the container image. We use it to copy all our source code from the newly created *src* folder, into a similarly named folder at the root of the container image.

❸ `WORKDIR` sets the current working directory for all subsequent commands. It's a bit like doing `cd /src`.

❹ Finally, the `CMD` instruction tells Docker which command you want it to run by default, when you start a container based on that image. The syntax is a bit like a Python list (although it's actually parsed as a JSON array, so you *have* to use double quotes).

It's probably worth just showing a directory tree, to make sure everything is in the right place, right? All our source code is in a folder called *src*, next to our `Dockerfile`:

```
.
├── Dockerfile
├── db.sqlite3
├── src
│   ├── functional_tests
│   │   ├── [...]
│   ├── lists
│   │   ├── [...]
│   ├── manage.py
│   └── superlists
│       ├── [...]
└── static
    └── [...]
```

Docker Build

You build an image with `docker build <path-containing-dockerfile>` and we'll use the `-t <tagname>` argument to "tag" our image with a memorable name.

It's typical to invoke `docker build` from the folder that contains your Dockerfile, so the last argument is usually `.`:

```
$ docker build -t superlists .
[+] Building 1.2s (8/8) FINISHED                              docker:default
 => [internal] load build definition from Dockerfile                   0.0s
 => => transferring dockerfile: 115B                                   0.0s
 => [internal] load .dockerignore                                      0.1s
 => => transferring context: 2B                                        0.0s
 => [internal] load metadata for docker.io/library/python:slim         3.4s
 => [internal] load build context                                      0.2s
 => => transferring context: 68.54kB                                   0.1s
 => [1/3] FROM docker.io/library/python:3.14-slim@sha256:858[...]      4.4s
 => => resolve docker.io/library/python:3.14-slim@sha256:858[...]      0.0s
 => => sha256:72ba3400286b233f3cce28e35841ed58c9e775d69cf11f[...]      0.0s
 => => sha256:3a72e7f66e827fbb943c494df71d2ae024d0b1db543bf6[...]      0.0s
 => => sha256:a7d9a0ac6293889b2e134861072f9099a06d78ca983d71[...]      0.5s
 => => sha256:426290db15737ca92fe1ee6ff4f450dd43dfc093e92804[...]      4.0s
 => => sha256:e8b685ab0b21e0c114aa94b28237721d66087c2bb53932[...]      0.5s
 => => sha256:85824326bc4ae27a1abb5bc0dd9e08847aa5fe73d8afb5[...]      0.0s
 => => extracting sha256:a7d9a0ac6293889b2e134861072f9099a06[...]      0.1s
 => => extracting sha256:426290db15737ca92fe1ee6ff4f450dd43d[...]      0.4s
 => => extracting sha256:e8b685ab0b21e0c114aa94b28237721d660[...]      0.0s
 => [internal] load build context                                      0.0s
 => => transferring context: 7.56kB                                    0.0s
 => [2/3] COPY src /src                                                0.2
 => [3/3] WORKDIR /src                                                 0.1s
 => exporting to image                                                 0.0s
 => => exporting layers                                                0.0s
 => => writing image sha256:7b8e1c9fa68e7bad7994fa41e2aca852ca79f01a   0.0s
 => => naming to docker.io/library/superlists                          0.0s
```

Now we can see our image in the list of Docker images on the system:

```
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
superlists   latest    522824a399de   2 minutes ago    164MB
[...]
```

If you see an error about `failed to solve / compute cache key` and `src: not found`, it may be because you saved the Dockerfile in the wrong place. Have another look at the directory tree from earlier.

Docker Run

Once you've built an image, you can run one or more containers based on that image, using `docker run`. What happens when we run ours?

```
$ docker run superlists
Traceback (most recent call last):
  File "/src/manage.py", line 11, in main
    from django.core.management import execute_from_command_line
ModuleNotFoundError: No module named 'django'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/src/manage.py", line 22, in <module>
    main()
    ~~~~^^
  File "/src/manage.py", line 13, in main
    raise ImportError(
    ...<3 lines>...
    ) from exc
ImportError: Couldn't import Django. Are you sure it's installed and available
on your PYTHONPATH environment variable? Did you forget to activate a virtual
environment?
```

Ah, we forgot that we need to install Django.

Installing Django in a Virtualenv in Our Container Image

Just like on our own machine, a virtualenv is useful in a deployed environment to make sure we have full control over the packages installed for a particular project.[6]

We can create a virtualenv in our Dockerfile just like we did on our own machine with `python -m venv`, and then we can use `pip install` to get Django:

Dockerfile (ch09l004)

```
FROM python:3.14-slim

RUN python -m venv /venv  ❶
ENV PATH="/venv/bin:$PATH"  ❷

RUN pip install "django<6" ❸

COPY src /src

WORKDIR /src

CMD ["python", "manage.py", "runserver"]
```

❶ Here's where we create our virtualenv. We use the `RUN` Dockerfile directive, which is how you run arbitrary shell commands as part of building your Docker image.

❷ You can't really "activate" a virtualenv inside a Dockerfile, so instead we change the system path so that the venv versions of `pip` and `python` become the default ones (this is actually one of the things that `activate` does, under the hood).

❸ We install Django with `pip install`, just like we do locally.

6 Even a completely fresh Linux install might have odd things installed in its system site packages. A virtualenv is a guaranteed clean slate.

Successful Run

Let's do the `build` and `run` in a single line. This is a pattern I used quite often when developing a Dockerfile, to be able to quickly rebuild and see the effect of a change:

```
$ docker build -t superlists . && docker run -it superlists
[+] Building 0.2s (11/11) FINISHED                              docker:default
[...]
 => [internal] load .dockerignore                                        0.1s
 => => transferring context: 2B                                          0.0s
 => [internal] load build definition from Dockerfile                     0.0s
 => => transferring dockerfile: 246B                                     0.0s
 => [internal] load metadata for docker.io/library/python:slim           0.0s
 => CACHED [1/5] FROM docker.io/library/python:slim                      0.0s
 => [internal] load build context                                        0.0s
 => => transferring context: 4.75kB                                      0.0s
 => [2/5] RUN python -m venv /venv                                       0.0s
 => [3/5] pip install "django<6"                                         0.0s
 => [4/5] COPY src /src                                                  0.0s
 => [5/5] WORKDIR /src                                                   0.0s
 => exporting to image                                                   0.0s
 => => exporting layers                                                  0.0s
 => => writing image sha256:[...]                                        0.0s
 => => naming to docker.io/library/superlists                            0.0s
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 19 unapplied migration(s). Your project may not [...]
[...]
Django version 5.2, using settings superlists.settings
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

OK, scanning through that, it looks like the server is running!

Make sure you use the `-it` flags to the Docker `run` command when running `runserver`, or any other tool that expects to be run in an interactive terminal session, otherwise you'll get strange behaviour, including not being able to interrupt the Docker process with Ctrl+C. See the following sidebar for an escape hatch.

How to Stop a Docker Container

If you've got a container that's "hanging" in a terminal window, you can stop it from another terminal.

The Docker daemon lets you list all the currently running containers with `docker ps`:

```
$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS
PORTS     NAMES
0818e1b8e9bf   superlists   "/bin/sh -c 'python …"   4 seconds ago   Up 4
seconds             hardcore_moore
```

This tells us a bit about each container, including a unique ID and a randomly-generated name (you can override that if you want to).

We can use the ID or the name to terminate the container with `docker stop`:[7]

```
$ docker stop 0818e1b8e9bf
0818e1b8e9bf
```

And if you go back to your other terminal window, you should find that the Docker process has been terminated.

Using the FT to Check That Our Container Works

Let's see what our FTs think about this Docker version of our site:

```
$ TEST_SERVER=localhost:8888 ./src/manage.py test src/functional_tests --failfast
[...]
selenium.common.exceptions.WebDriverException: Message: Reached error page: abo
ut:neterror?e=connectionFailure&u=http%3A//localhost%3A8888/[...]
```

What's going on here? Time for a little debugging.

7 There is also a `docker kill` if you're in a hurry. But `docker stop` will send a `SIGKILL` if its initial `SIGTERM` doesn't work within a certain timeout (more info in the Docker docs).

Debugging Container Networking Problems

First, let's try and take a look ourselves, in our browser, by going to *http://localhost:8888/*, as in Figure 9-3.

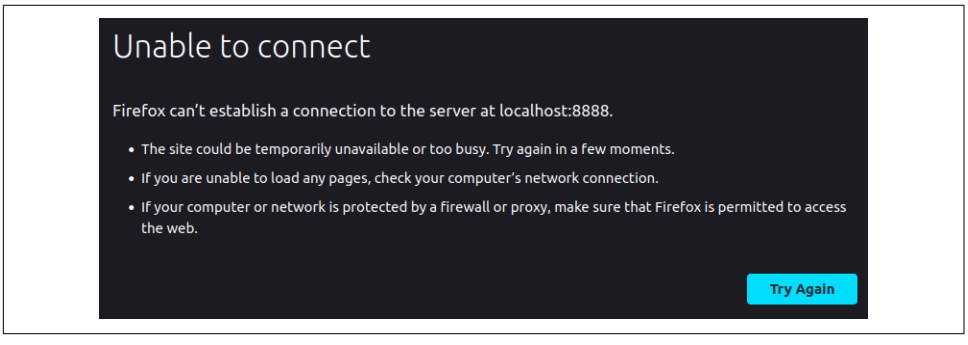


Figure 9-3. Cannot connect on that port

Now, let's take another look at the output from our `docker run`. Here's what appeared right at the end:

```
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Aha! We notice that we're using the wrong port, the default `8000` instead of the `8888` that we specified in the `TEST_SERVER` environment variable (or, "env var").

Let's fix that by amending the `CMD` instruction in the Dockerfile:

Dockerfile (ch09l005)

```
[...]
WORKDIR /src

CMD ["python", "manage.py", "runserver", "8888"]
```

Ctrl+C the current Dockerized container process if it's still running in your terminal, then give it another `build && run`:

```
$ docker build -t superlists . && docker run -it superlists
[...]
Starting development server at http://127.0.0.1:8888/
```

Debugging Web Server Connectivity with curl

A quick run of the FT or check in our browser will show us that nope, that doesn't work either. Let's try an even lower-level smoke test, the traditional Unix utility `curl`. It's a command-line tool for making HTTP requests.[8] Try it on your own computer first:

```
$ curl -iv localhost:8888
*   Trying 127.0.0.1:8888...
* connect to 127.0.0.1 port 8888 [...]
*   Trying [::1]:8888...
* connect to ::1 port 8888 [...]
* Failed to connect to localhost port 8888 after 0 ms: [...]
* Closing connection
[...]
curl: (7) Failed to connect to localhost port 8888 after 0 ms: [...]
```

The `-iv` flag to `curl` is useful for debugging. It prints verbose output, as well as full HTTP headers.

Running Code "Inside" the Container with docker exec

So, we can't see Django running on port `8888` when we're *outside* the container. What do we see if we run things from *inside* the container?

We can use `docker exec` to run commands inside a running container. First, we need to get the name or ID of the container:

```
$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED          STATUS
PORTS     NAMES
5ed84681fdf8   superlists   "/bin/sh -c 'python …"   12 minutes ago   Up 12
minutes             trusting_wu
```

Your values for `CONTAINER_ID` and `NAMES` will be different from mine, because they're randomly generated. But make a note of one or the other, and then run `docker exec -it <container-id> bash`. On most platforms, you can use tab completion for the container ID or name.

8 `curl` can do FTP (File Transfer Protocol) and many other types of network requests too! Check out the `curl` manual.

Let's try it now. Notice that the shell prompt will change from your default Bash prompt to `root@container-id`. Watch out for those in future listings, so that you can be sure of what's being run inside versus outside containers.

```
$ docker exec -it container-id-or-name bash
root@5ed84681fdf8:/src# apt-get update && apt-get install -y curl
Get:1 http://deb.debian.org/debian bookworm InRelease [151 kB]
Get:2 http://deb.debian.org/debian bookworm-updates InRelease [52.1 kB]
[...]
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  libbrotli1 libcurl4 libldap-2.5-0 libldap-common libnghttp2-14 libpsl5
[...]
root@5ed84681fdf8:/src# curl -iv http://localhost:8888
*   Trying [...]
* Connected to localhost [...]
> GET / HTTP/1.1
> Host: localhost:8888
> User-Agent: curl/8.6.0
> Accept: */*
>
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
[...]
<!doctype html>
<html lang="en">

  <head>
    <title>To-Do lists</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link href="/static/bootstrap/css/bootstrap.min.css" rel="stylesheet">
  </head>

  <body>
    [...]
  </body>

</html>
```

Use Ctrl+D to exit from the `docker exec bash` shell inside the container.

That's definitely some HTML! And the `<title>To-Do lists</title>` looks like it's our HTML, too.

So, we can see Django is serving our site *inside* the container. Why can't we see it *outside*?

Docker Port Mapping

The (highly, highly recommend) PythonSpeed guide to Docker's very first section is called Connection refused?, so I'll refer you there once again for an *excellent*, detailed explanation.

But in short: Docker runs in its own little world; specifically, it has its own little network, so the ports *inside* the container are different from the ports *outside* the container, the ones we can see on our host machine.

So, we need to tell Docker to connect the internal ports to the outside ones—to "publish" or "map" them, in Docker terminology.

`docker run` takes a `-p` argument, with the syntax `OUTSIDE:INSIDE`. So, you can actually map a different port number on the inside and outside. But we're just mapping `8888` to `8888`, and that will look like this:

```
$ docker build -t superlists . && docker run -p 8888:8888 -it superlists
```

Now that will *change* the error we see, but only quite subtly (see Figure 9-4).[9] Things clearly aren't working yet.

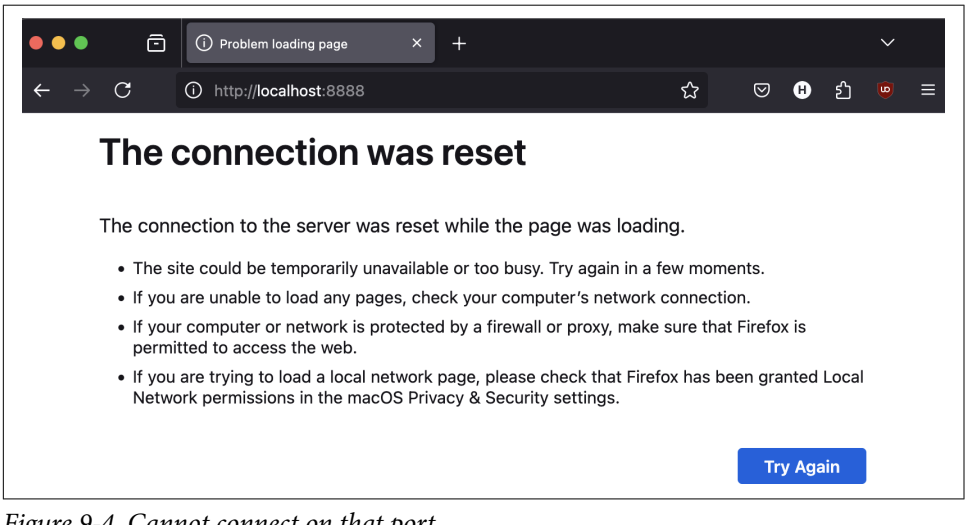


Figure 9-4. Cannot connect on that port

9 Tip: If you use Chrome as your web browser, its error is something like "localhost didn't send any data. ERR_EMPTY_RESPONSE".

Similarly, if you try our `curl -iv` (outside the container) once again, you'll see the error has changed from "Failed to connect", to "Empty reply":

```
$ curl -iv localhost:8888
*   Trying [...]
* Connected to localhost (127.0.0.1) port 8888
> GET / HTTP/1.1
> Host: localhost:8888
> User-Agent: curl/8.6.0
> Accept: */*
[...]
* Empty reply from server
* Closing connection
curl: (52) Empty reply from server
```

Depending on your system, instead of `(52) Empty reply from server`, you might see `(56) Recv failure: Connection reset by peer`. They mean the same thing: we can connect but we don't get a response.

Essential Googling the Error Message

The need to map ports and the `-p` argument to `docker run` are something you just pick up, fairly early on in learning Docker. But the next debugging step is quite a bit more obscure—although admittedly Itamar does address it in his Docker networking article (did I already mention how excellent it is?).

But if we haven't read that, we can always resort to the tried and tested "Googling the error message" technique instead (Figure 9-5).

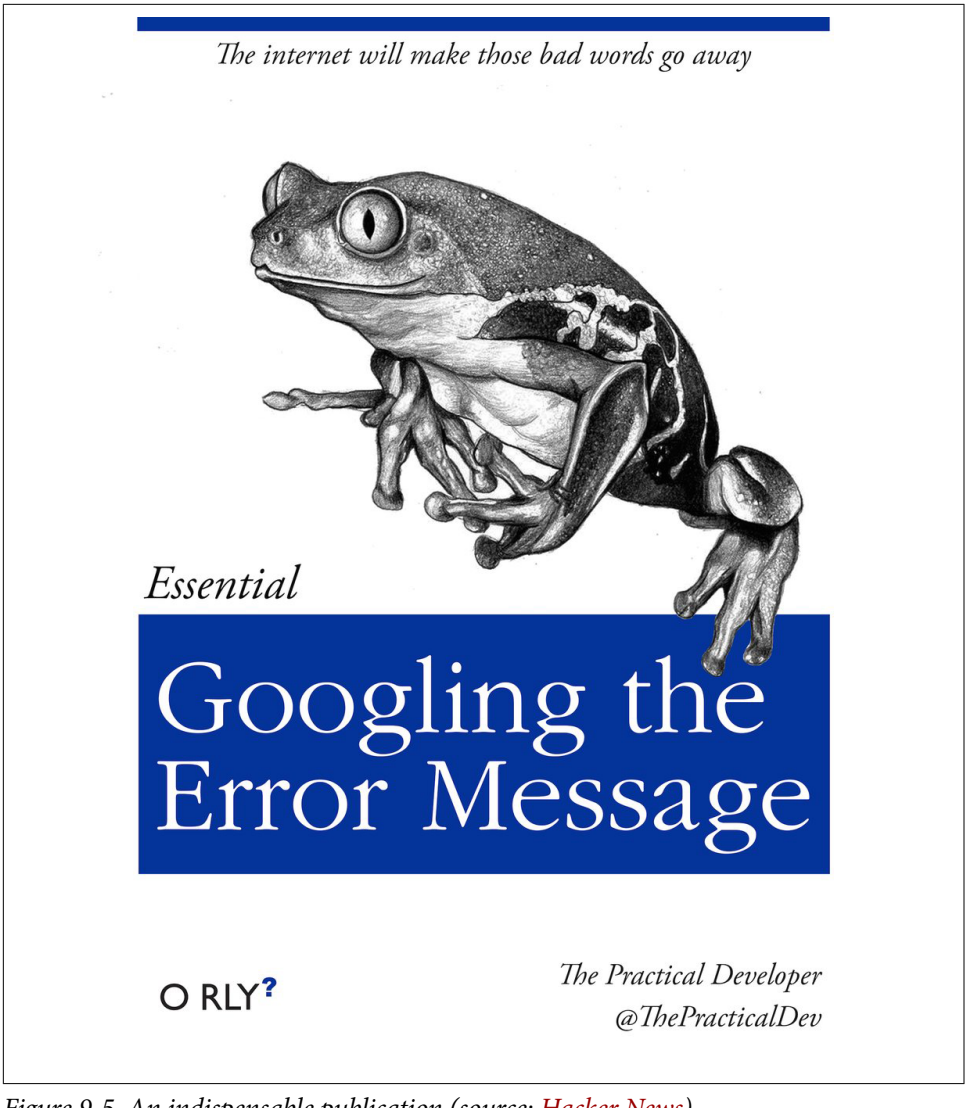


Figure 9-5. An indispensable publication (source: Hacker News)

Everyone's search results are a little different, and mine are perhaps shaped by years of working with Docker and Django, but I found the answer in my very first result (see Figure 9-6), when I searched for "cannot access Django runserver inside Docker". The result was was a Stack Overflow post, saying something about needing to specify `0.0.0.0` as the IP address.[10]

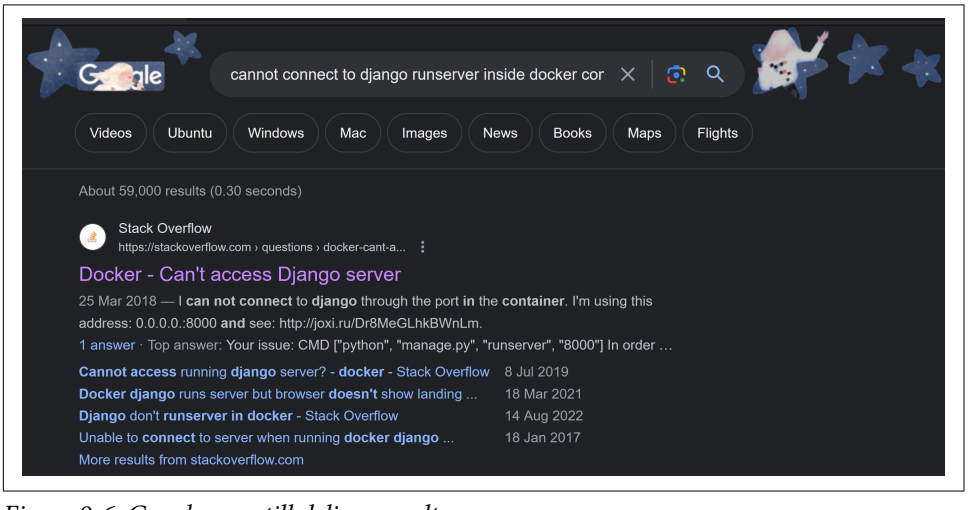


Figure 9-6. Google can still deliver results

We're nearing the edges of my understanding of Docker now, but as I understand it, `runserver` binds to `127.0.0.1` by default. However, that IP address doesn't correspond to a network adapter *inside* the container, which is actually connected to the outside world via the port mapping we defined earlier.

10 Kids these days will probably ask an AI right? I have to say, I tried it out, with the prompt being "I'm trying to run Django inside a Docker container, and I've mapped port 8888, but I still can't connect. Can you suggest what the problem might be?", and it come up with a pretty good answer.

The long and short of it is that we need use the long-form `ipaddr:port` version of the `runserver` command, using the magic "wildcard" IP address, `0.0.0.0`:

Dockerfile (ch09l007)

```
[...]
WORKDIR /src

CMD ["python", "manage.py", "runserver", "0.0.0.0:8888"]
```

Rebuild and rerun your server, and if you have eagle eyes, you'll spot it's binding to `0.0.0.0` instead of `127.0.0.1`:

```
$ docker build -t superlists . && docker run -p 8888:8888 -it superlists
[...]
Starting development server at http://0.0.0.0:8888/
```

We can verify it's working with `curl`:

```
$ curl -iv localhost:8888
*   Trying [...]
* Connected to localhost [...]
[...]

  </body>

</html>
```

Looking good!

On Debugging

Let me let you in on a little secret: I'm actually not that good at debugging. We all have our psychological strengths and weaknesses, and one of my weaknesses is that when I run into a problem that I can't see an obvious solution to, I want to throw up my hands way too soon and say "well, this is hopeless; it can't be fixed", and give up.

Thankfully I have had some good role models over the years who are much better at it than me (hi, Glenn!). Debugging needs the patience and tenacity of a bloodhound. If at first you don't succeed, you need to systematically rule out options, check your assumptions, eliminate various aspects of the problem, simplify things down, and find the parts that do and don't work, until you eventually find the cause.

It might seems hopeless at first! But you usually get there eventually.

Database Migrations

A quick visual inspection confirms—the site is up (Figure 9-7)!

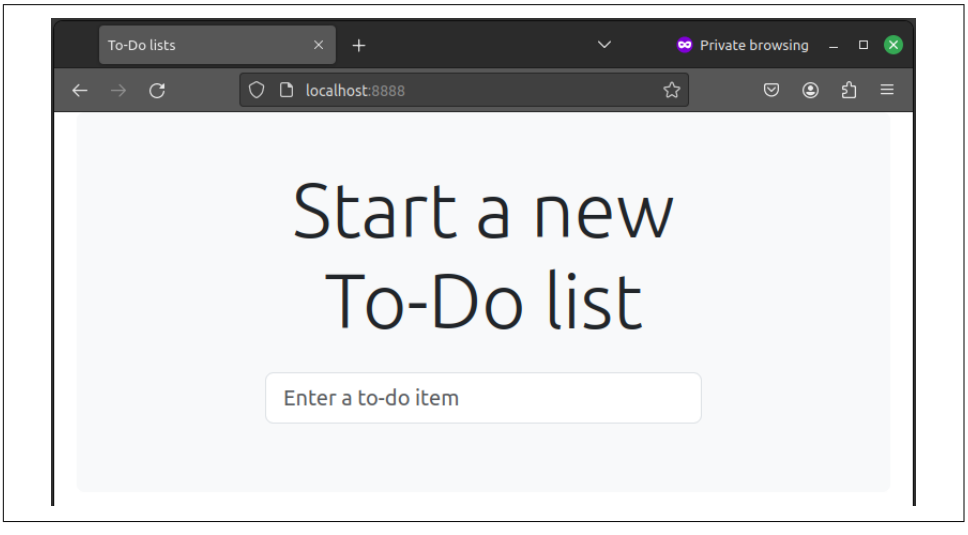


Figure 9-7. The site in Docker is up!

Let's see what our functional tests say:

```
$ TEST_SERVER=localhost:8888 ./src/manage.py test src/functional_tests --failfast
[...]
E
======================================================================
ERROR: test_can_start_a_todo_list
(functional_tests.tests.NewVisitorTest.test_can_start_a_todo_list)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/functional_tests/tests.py", line 56, in
test_can_start_a_todo_list
    self.wait_for_row_in_list_table("1: Buy peacock feathers")
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "...goat-book/src/functional_tests/tests.py", line 26, in
wait_for_row_in_list_table
    table = self.browser.find_element(By.ID, "id_list_table")
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; For documentation [...]
```

Although the FTs can connect happily and interact with our site, they are failing as soon as they try to submit a new item.

You might have spotted the yellow Django debug page (Figure 9-8) telling us why. It's because we haven't set up the database (which, as you may remember, we highlighted as one of the "danger areas" of deployment).

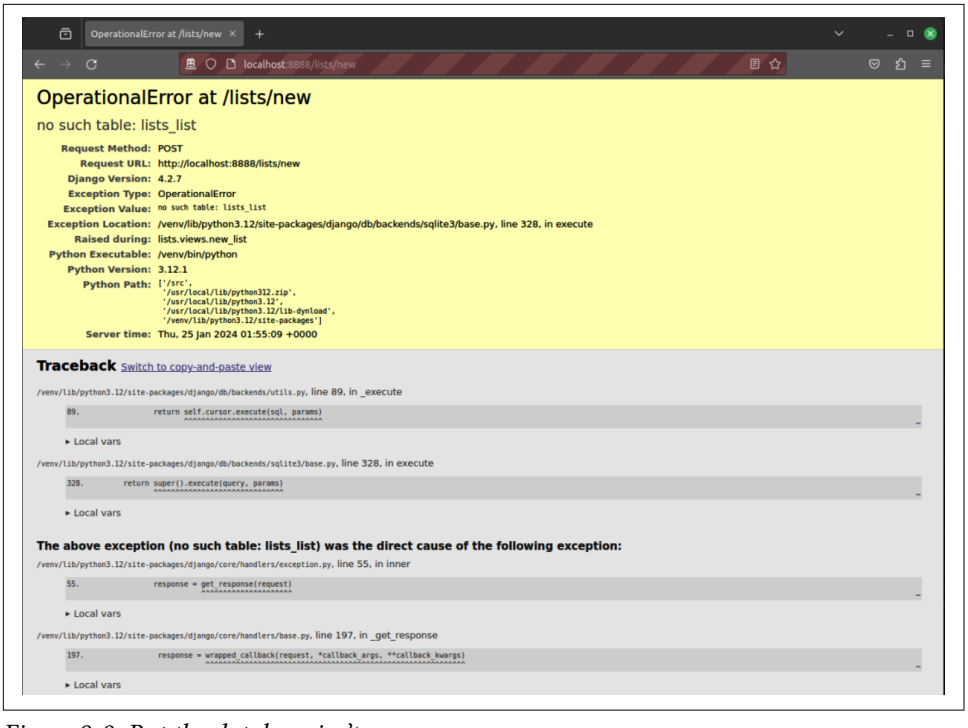


Figure 9-8. But the database isn't

The tests saved us from potential embarrassment there. The site *looked* fine when we loaded its front page. If we'd been a little hasty and only tested manually, we might have thought we were done, and it would have been the first users that discovered that nasty Django debug page. Okay, slight exaggeration for effect—maybe we *would* have checked, but what happens as the site gets bigger and more complex? You can't check everything. The tests can.

To be fair, if you look back through the `runserver` command output each time we've been starting our container, you'll see it's been warning us about this issue:

```
You have 19 unapplied migration(s). Your project may not work properly until
you apply the migrations for app(s): auth, contenttypes, lists, sessions.
Run 'python manage.py migrate' to apply them.
```

If you don't see this error, it's because your *src* folder had the database file in it, unlike mine. For the sake of argument, run `rm src/db.sqlite3` and rerun the build and run commands, and you should be able to reproduce the error. I promise it's instructive!

Should We Run migrate Inside the Dockerfile? No.

So, should we include `manage.py migrate` in our Dockerfile?

If you try it, you'll find it certainly *seems* to fix the problem:

Dockerfile (ch09l008)

```
[...]
WORKDIR /src

RUN python manage.py migrate --noinput  ❶

CMD ["python", "manage.py", "runserver", "0.0.0.0:8888"]
```

❶ We run `migrate` using the `--noinput` argument to suppress any little "are you sure" prompts.

If we rebuild the image…

```
$ docker build -t superlists . && docker run -p 8888:8888 -it superlists
[...]
Starting development server at http://0.0.0.0:8888/
```

…and try our FTs again, they all pass!

```
$ TEST_SERVER=localhost:8888 ./src/manage.py test src/functional_tests --failfast
[...]
...
 ----------------------------------------------------------------------
Ran 3 tests in 26.965s

OK
```

The problem is that this saves our database file into our system image, which is not what we want, because the system image is meant to be something fixed and stateless (whereas the database is living, stateful data that should change over time).

What Would Happen if We Kept the Database File in the Image

You can try this as a little experiment. Assuming you've got the `manage.py migrate` line in your Dockerfile:

1. Create a new to-do list and keep a note of its URL (e.g., at *http://localhost:8888/lists/1*).
2. Now, `docker stop` your container, and rebuild a new one with the same `build && run` command we used earlier.
3. Go back and try to retrieve your old list. It's gone!

This is because rebuilding the image will give us a brand new database each time.

What we actually want is for our database storage to be "outside" the container somehow, so it can persist between different versions of our Docker image.

Mounting Files Inside the Container

We want the database on the server to be totally separate data from the data in the system image. In most deployments, you'd probably be talking to a separate database server, like PostgreSQL. For the purposes of this book, the easiest analogy for a database that's "outside" our container is to access the database from the filesystem outside the container.

That also gives us a convenient excuse to talk about mounting files in Docker, which is a very Useful Thing to be Able to Do™.

First, let's revert our change:

Dockerfile (ch09l009)

```
[...]
COPY src /src

WORKDIR /src

CMD ["python", "manage.py", "runserver", "0.0.0.0:8888"]
```

Then, let's make sure we *do* have the database on our local filesystem, by running `migrate` (when we moved everything into *./src*, we left the database file behind):

```
$ ./src/manage.py migrate --noinput
Operations to perform:
  Apply all migrations: auth, contenttypes, lists, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
[...]
  Applying sessions.0001_initial... OK
```

Let's make sure to *.gitignore* the new location of the database file, and we'll also use a file called *.dockerignore* to make sure we can't copy our local dev database into our Docker image during Docker builds:

```
$ echo src/db.sqlite3 >> .gitignore
$ echo src/db.sqlite3 >> .dockerignore
```

Now we rebuild, and try mounting our database file. The extra flag to add to the Docker run command is `--mount`, where we specify `type=bind`, the `source` path on our machine,[11] and the `target` path *inside* the container:

```
$ docker build -t superlists . && docker run \
  -p 8888:8888 \
  --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
  -it superlists
```

You're likely to come across the old syntax for mounts, which was `-v`. One of the advantages of the new `--mount` version is that it will fail hard if the path you're trying to mount does not exist—it says something like `bind source path does not exist`. This avoids a lot of pain (ask me how I know this).

```
$ TEST_SERVER=localhost:8888 ./src/manage.py test src/functional_tests --failfast
[...]
...
 ---------------------------------------------------------------------
Ran 3 tests in 26.965s

OK
```

AMAZING, IT ACTUALLY WORKSSSSSSSS.

Ahem, that's definitely good enough for now! Let's commit:

```
$ git add -A .  # add Dockerfile, .dockerignore, .gitignore
$ git commit -am"First cut of a Dockerfile"
```

11 If you're wondering about the `$PWD` in the listing, it's a special environment variable that represents the current directory. The initials echo the `pwd` command, which stands for "print working directory". Docker requires mount paths to be absolute paths.

Phew. Well, it took a bit of hacking about, but now we can be reassured that the basic Docker plumbing works. Notice that the FT was able to guide us incrementally towards a working config, and spot problems early on (like the missing database).

But we really can't be using the Django dev server in production, or running on port `8888` forever. In the next chapter, we'll make our hacky image more production-ready.

But first, time for a well-earned tea break I think, and perhaps a chocolate biscuit.

Docker Recap

Docker lets us reproduce a server environment on our own machine
: For developers, ops and infra work is always "fun", by which I mean a process full of fear, uncertainty, and surprises—and painfully slow too. Docker helps to minimise this pain by giving us a mini server on our own machine, which we can try things out with and get feedback quickly, as well as enable us to work in small steps.

`docker build && docker run`
: We've learned the core tools for working with Docker. The Dockerfile specifies our image, `docker build` builds it, and `docker run` runs it. `build && run` together give us a "start again from scratch" cycle, which we use every time we make a code change in *src*, or a change in the Dockerfile.[12]

Debugging network issues
: We've seen how to use `curl` both outside and inside the container with `docker exec`. We've also seen the `-p` argument to bind ports inside and outside, and the idea of needing to bind to `0.0.0.0`.

Mounting files
: We've also had a brief intro to mounting files from outside the container, into the inside. It's an insight into the difference between the "stateless" system image, and the stateful world outside of Docker.

12 There's a common pattern of mounting the whole *src* folder into your Docker containers in local dev. It means you don't need to rebuild for every source code change. I didn't wan't to introduce that here because it also leads to subtle behaviours that can be hard to wrap your head around, like the *db.sqlite3* file being shared with the container. For this book, the `build && run` cycle is fast enough, but by all means try out mounting *src* in your own projects.

CHAPTER 10

Making Our App Production-Ready

Our container is working fine but it's not production-ready. Let's try to get it there, using the tests to keep us safe.

In a way we're applying the red/green/refactor cycle to our productionisation process. Our hacky container config got us to green, and now we're going to refactor, working incrementally (just as we would while coding), trying to move from working state to working state, and using the FTs to detect any regressions.

What We Need to Do

What's wrong with our hacky container image? A few things: first, we need to host our app on the "normal" port `80` so that people can access it using a regular URL.

Perhaps more importantly, we shouldn't use the Django dev server for production; it's not designed for real-life workloads. Instead, we'll use the popular Gunicorn Python WSGI HTTP server.

Django's `runserver` is built and optimised for local development and debugging. It's designed to handle one user at a time; it handles automatic reloading upon saving of the source code, but it isn't optimised for performance, nor has it been hardened against security vulnerabilities.

In addition, several options in *settings.py* are currently unacceptable. `DEBUG=True` is strongly discouraged for production, we'll want to set a unique `SECRET_KEY` and, as we'll see, other things will come up.

`DEBUG=True` is considered a security risk, because the Django debug page will display sensitive information like the values of variables, and most of the settings in *settings.py*.

Let's go through and see if we can fix things one by one.

Switching to Gunicorn

Do you know why the Django mascot is a pony? The story is that Django comes with so many things you want: an ORM, all sorts of middleware, the admin site…"What else do you want, a pony?" Well, Gunicorn stands for "Green Unicorn", which I guess is what you'd want next if you already had a pony…

We'll need to first install Gunicorn into our container, and then use it instead of `runserver`:

```
$ python -m pip install gunicorn
Collecting gunicorn
[...]
Successfully installed gunicorn-2[...]
```

Gunicorn will need to know a path to a "WSGI server"[1] which is usually a function called `application`. Django provides one in *superlists/wsgi.py*. Let's change the command that our image runs:

Dockerfile (ch10l001)

```
[...]
RUN pip install "django<6" gunicorn  ❶

COPY src /src

WORKDIR /src

CMD ["gunicorn", "--bind", ":8888", "superlists.wsgi:application"]  ❷
```

❶ Installation is a standard `pip install`.

❷ Gunicorn has its own command line, `gunicorn`. Here's where we invoke it, including telling it which port to use, and supplying the dot-notation path to the WSGI server provided by Django.

1 WSGI stands for Web Server Gateway Interface and it's the protocol for communication between a web server and a Python web application. Gunicorn is a web server that uses WSGI to interact with Django, and so is the web server you get from `runserver`.

As in the previous chapter, we can use the `docker build && docker run` pattern to try out our changes by rebuilding and rerunning our container:

```
$ docker build -t superlists . && docker run \
  -p 8888:8888 \
  --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
  -it superlists
```

If you see an error saying `Bind for 0.0.0.0:8888 failed: port is already allocated.`, it'll be because you still have a container running from the previous chapter. Do you remember how to use `docker ps` and `docker stop`? If not, have another look at "How to Stop a Docker Container" on page 208.

The FTs Catch a Problem with Static Files

As we run the FTs, you'll see them warning us of a problem, once again. The test for adding list items passes happily, but the test for layout and styling fails. Good job, tests!

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]
AssertionError: 102.5 != 512 within 10 delta (409.5 difference)
FAILED (failures=1)
```

And indeed, if you take a look at the site, you'll find the CSS is all broken, as in Figure 10-1.

The reason that we have no CSS is that although the Django dev server will serve static files magically for you, Gunicorn doesn't.

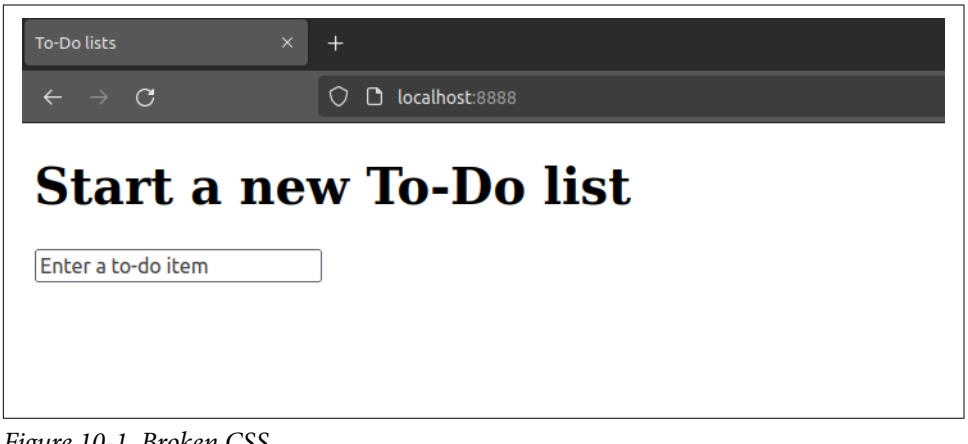


Figure 10-1. Broken CSS

One step forwards, one step backwards, but once again we've identified the problem nice and early. Moving on!

Serving Static Files with WhiteNoise

Serving static files is very different from serving dynamically rendered content from Python and Django. There are many ways to serve them in production: you can use a web server like nginx, or a content delivery network (CDN) like Amazon S3. But in our case, the most straightforward thing to do is to use WhiteNoise, a Python library expressly designed for serving static[2] files from Python.

First, we install WhiteNoise into our local environment:

```
pip install whitenoise
```

Then we tell Django to enable it, in *settings.py*[3]:

src/superlists/settings.py (ch10l002)

```
MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    [...]
```

And then we need to add it to our `pip installs` in the Dockerfile:

Dockerfile (ch10l003)

```
RUN pip install "django<6" gunicorn whitenoise
```

This manual list of `pip installs` is getting a little fiddly! We'll come back to that in a moment. First let's rebuild and try rerunning our FTs:

```
$ docker build -t superlists . && docker run \
  -p 8888:8888 \
  --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
  -it superlists
```

And if you take another manual look at your site, things should look much healthier.

2 Believe it or not, this pun didn't actually hit me until I was rewriting this chapter. For 10 years, it was right under my nose. I think that makes it funnier actually.

3 Find out more about Django middleware in the docs.

Let's rerun our FTs to confirm:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]

...
 ----------------------------------------------------------------------
Ran 3 tests in 10.718s

OK
```

Phew. Let's commit that:

```
$ git commit -am"Switch to Gunicorn and Whitenoise"
```

Using requirements.txt

Let's deal with that fiddly list of `pip install`s.

To reproduce our local virtualenv, rather than just manually `pip install`ing things one by one and having to remember to sync things between local dev and Docker, we can "save" the list of packages we're using by creating a *requirements.txt* file.[4]

The `pip freeze` command will show us everything that's installed in our virtualenv at the moment:

```
$ pip freeze
asgiref==3.8.1
attrs==25.3.0
certifi==2025.4.26
Django==5.2.3
gunicorn==23.0.0
h11==0.16.0
idna==3.10
outcome==1.3.0.post0
packaging==25.0
PySocks==1.7.1
selenium==4.31.0
sniffio==1.3.1
sortedcontainers==2.4.0
sqlparse==0.5.3
trio==0.30.0
trio-websocket==0.12.2
typing_extensions==4.13.2
urllib3==2.4.0
websocket-client==1.8.0
whitenoise==6.11.0
wsproto==1.2.0
```

4 There are many other dependency management tools these days so *requirements.txt* is not the only way to do it, although it is one of the oldest and best established. As you continue your Python adventures, I'm sure you'll come across many others.

That shows *all* the packages in our virtualenv, along with their version numbers. Let's pull out just the "top-level" dependencies—Django, Gunicorn, and WhiteNoise:

```
$ pip freeze | grep -i django
Django==5.2[...]

$ pip freeze | grep -i django >> requirements.txt
$ pip freeze | grep -i gunicorn >> requirements.txt
$ pip freeze | grep -i whitenoise >> requirements.txt
```

That should give us a *requirements.txt* file that looks like this:

requirements.txt (ch10l004)

```
django==5.2.3
gunicorn==23.0.0
whitenoise==6.11.0
```

Let's try it out! To install things from a *requirements.txt* file, you use the `-r` flag, like this:

```
$ pip install -r requirements.txt
Requirement already satisfied: Django==5.2.[...]
./.venv/lib/python3.14/site-packages (from -r requirements.txt (line 1))
(5.2.[...]
Requirement already satisfied: gunicorn==23.0.0 in
./.venv/lib/python3.14/site-packages (from -r requirements.txt (line 2))
(23.0.0)
Requirement already satisfied: whitenoise==6.11.0 in
./.venv/lib/python3.14/site-packages (from -r requirements.txt (line 3))
(6.11.0)
Requirement already satisfied: asgiref[...]
Requirement already satisfied: sqlparse[...]
[...]
```

As you can see, it's a no-op because we already have everything installed. That's expected!

Forgetting the `-r` and running `pip install requirements.txt` is such a common error, that I recommend you do it *right now* and get familiar with the error message (which is thankfully much more helpful than it used to be). It's a mistake I still make, *all the time*.

Anyway, that's a good first version of a requirements file. Let's commit it:

```
$ git add requirements.txt
$ git commit -m "Add a requirements.txt with Django, gunicorn and whitenoise"
```

Dev Dependencies, Transitive Dependencies, and Lockfiles

You may be wondering why we didn't add our other key dependency, Selenium, to our requirements. Or you might be wondering why we didn't just add *all* the dependencies, including the "transitive" ones (e.g., Django has its own dependencies like `asgiref` and `sqlparse`, etc.).

As always, I have to gloss over some nuance and trade-offs, but the short answer is: Selenium is only a dependency for the tests, not the application code; we're never going to run the tests directly on our production servers.[5] As for transitive dependencies, they're fiddly to manage without bringing in more tools, and I didn't want to do that for this book.

When you have a moment, you should probably to do some further reading on "lockfiles", *pyproject.toml*, hard pinning versus soft pinning, and immediate versus transitive dependencies.

If I absolutely *had* to recommend a Python dependency management tool, it would be pip-tools, which is a fairly minimal one.

Now let's see how we use that requirements file in our Dockerfile:

Dockerfile (ch10l005)

```
FROM python:3.14-slim

RUN python -m venv /venv
ENV PATH="/venv/bin:$PATH"

COPY requirements.txt /tmp/requirements.txt  ❶
RUN pip install -r /tmp/requirements.txt  ❷

COPY src /src

WORKDIR /src

CMD ["gunicorn", "--bind", ":8888", "superlists.wsgi:application"]
```

❶ We copy our requirements file in, just like the *src* folder.

❷ Now instead of just installing Django, we install all our dependencies using `pip install -r`.

5 Some people like to separate out test or "dev" dependencies into a separate requirements file called *requirements.dev.txt*, for example. For the record, I think this is a good idea, I just didn't want to add yet another concept to the book.

Let's build and run:

```
$ docker build -t superlists . && docker run \
  -p 8888:8888 \
  --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
  -it superlists
```

And then test to check everything still works:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]

OK
```

Hooray. That's a commit!

```
$ git commit -am "Use requirements.txt in Dockerfile"
```

Using Environment Variables to Adjust Settings for Production

We know there are several things in *settings.py* that we want to change for production:

- `DEBUG` mode is all very well for hacking about on your own server, but it isn't secure. For example, exposing raw tracebacks to the world is a bad idea.
- `SECRET_KEY` is used by Django for some of its crypto—things like cookies and CSRF protection. It's good practice to make sure the secret key in production is different from the one in your source code repo, because that code might be visible to strangers. We'll want to generate a new, random one but then keep it the same for the foreseeable future (find out more in the Django docs).

Development, staging, and production sites always have some differences in their configuration. Environment variables are a good place to store those different settings.[6]

6 The approach of using environment variables for configuration was originally published by "The 12-Factor App" manifesto. Another common way of handling this is to have different versions of *settings.py* for dev and prod. That can work fine too, but it can get confusing to manage. Environment variables also have the advantage of working for non-Django stuff too.

Setting DEBUG=True and SECRET_KEY

There are lots of ways you might set these settings.

What I propose may seem a little fiddly, but I'll provide a little justification for each choice. Let them be an inspiration (but not a template) for your own choices!

Note that this `if` statement replaces the `DEBUG` and `SECRET_KEY` lines that are included by default in the *settings.py* file:

src/superlists/settings.py (ch10l006)

```
import os
[...]

# SECURITY WARNING: don't run with debug turned on in production!
if "DJANGO_DEBUG_FALSE" in os.environ:  ❶
    DEBUG = False
    SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]  ❷
else:
    DEBUG = True  ❸
    SECRET_KEY = "insecure-key-for-dev"
```

❶ We say we'll use an environment variable called `DJANGO_DEBUG_FALSE` to switch debug mode off and, in effect, require production settings (it doesn't matter what we set it to, just that it's there).

❷ And now we say that, if debug mode is off, we *require* the `SECRET_KEY` to be set by a second environment variable.

❸ Otherwise we fall back to the insecure, debug mode settings that are useful for dev.

The end result is that you don't need to set any env vars for dev, but production needs both to be set explicitly, and it will error if any are missing. I think this gives us a little bit of protection against accidentally forgetting to set one.

Better to fail hard than allow a typo in an environment variable name to leave you running with insecure settings.

Setting Environment Variables Inside the Dockerfile

Now let's set that environment variable in our Dockerfile using the `ENV` directive:

Dockerfile (ch10l007)

```
WORKDIR /src

ENV DJANGO_DEBUG_FALSE=1

CMD ["gunicorn", "--bind", ":8888", "superlists.wsgi:application"]
```

And try it out…

```
$ docker build -t superlists . && docker run \
  -p 8888:8888 \
  --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
  -it superlists

[...]
  File "/src/superlists/settings.py", line 23, in <module>
    SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]
                 ~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^
[...]
KeyError: 'DJANGO_SECRET_KEY'
```

Oops. I forgot to set said secret key env var, mere seconds after having dreamt it up!

Setting Environment Variables at the Docker Command Line

We've said we can't keep the secret key in our source code, so the Dockerfile isn't an option; where else can we put it?

For now, we can set it at the command line using the `-e` flag for `docker run`:

```
$ docker build -t superlists . && docker run \
  -p 8888:8888 \
  --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
  -e DJANGO_SECRET_KEY=sekrit \
  -it superlists
```

With that running, we can use our FT again to see if we're back to a working state.

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]
AssertionError: 'To-Do' not found in 'Bad Request (400)'
```

The eagle-eyed might spot a message saying `UserWarning: No directory at: /src/static/`. That's a little clue about a problem with static files, which we're going to deal with shortly. Let's deal with this 400 issue first.

ALLOWED_HOSTS Is Required When Debug Mode Is Turned Off

It's not quite working yet (see Figure 10-2)! Let's take a look manually.

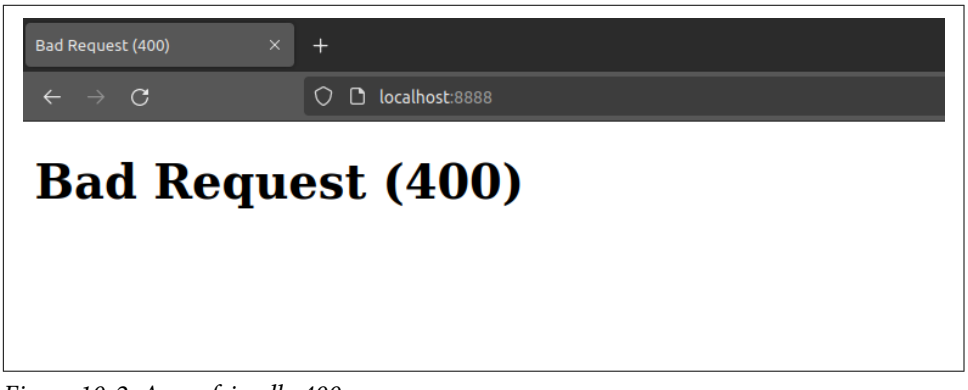


Figure 10-2. An unfriendly 400 error

We've set our two environment variables, but doing so seems to have broken things. However, once again, by running our FTs frequently, we're able to identify the problem early, before we've changed too many things at the same time. We've only changed two settings—which one might be at fault?

Let's use the "Googling the error message" technique again, with the search terms "Django debug false" and "400 bad request".

Well, the very first link in my search results was Stack Overflow suggesting that a 400 error is usually to do with `ALLOWED_HOSTS`. And the second was the official Django docs, which takes a bit more scrolling, but confirms it (see Figure 10-3).

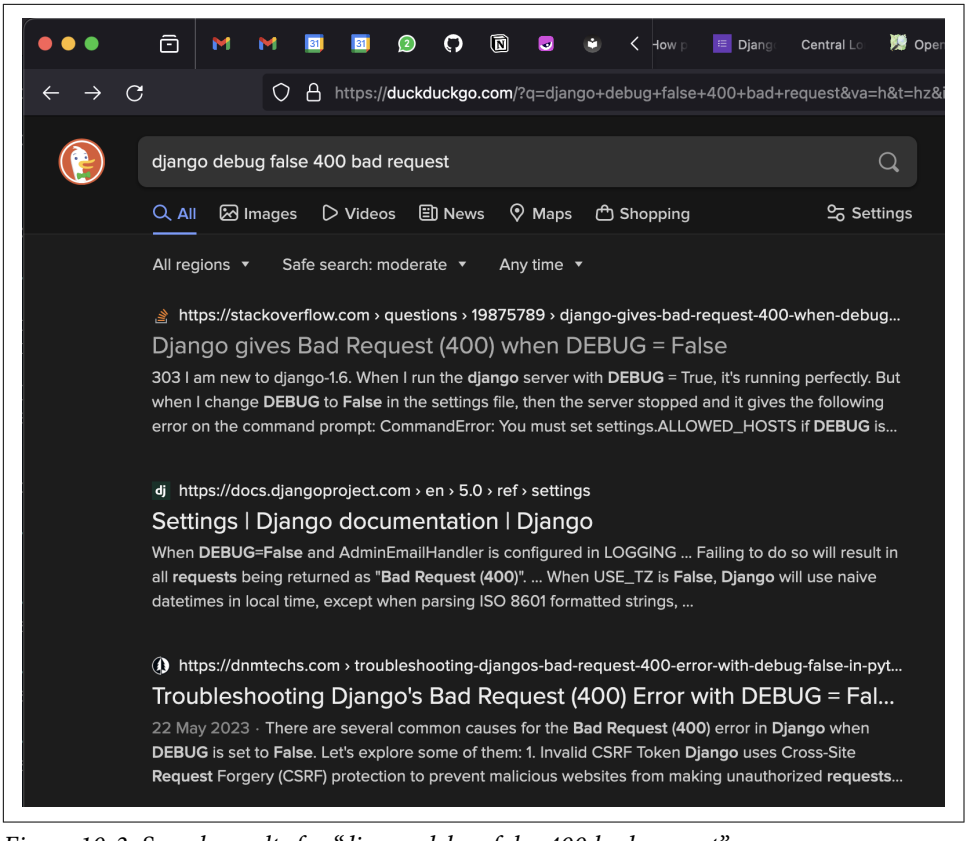


Figure 10-3. Search results for "django debug false 400 bad request"

`ALLOWED_HOSTS` is a security setting designed to reject requests that are likely to be forged, broken, or malicious because they don't appear to be asking for your site.[7]

When `DEBUG=True`, `ALLOWED_HOSTS` effectively allows *localhost* (our own machine) by default, so that's why it was working OK until now.

There's more information in the Django docs.

7 HTTP requests contain the address they were intended for in a header called "host".

The upshot is that we need to adjust `ALLOWED_HOSTS` in *settings.py*. Let's use another environment variable for that:

src/superlists/settings.py (ch10l008)

```
if "DJANGO_DEBUG_FALSE" in os.environ:
    DEBUG = False
    SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]
    ALLOWED_HOSTS = [os.environ["DJANGO_ALLOWED_HOST"]]
else:
    DEBUG = True
    SECRET_KEY = "insecure-key-for-dev"
    ALLOWED_HOSTS = []
```

This is a setting that we want to change, depending on whether our Docker image is running locally or on a server, so we'll use the `-e` flag again:

```
$ docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -it superlists
```

Collectstatic Is Required when Debug Is Turned Off

An FT run (or just looking at the site) reveals that we've had a regression in our static files:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]
AssertionError: 102.5 != 512 within 10 delta (409.5 difference)
FAILED (failures=1)
```

And you might have seen this warning message in the `docker run` output:

```
/venv/lib/python3.14/site-packages/django/core/handlers/base.py:61:
UserWarning: No directory at: /src/static/
  mw_instance = middleware(adapted_handler)
```

We saw this at the beginning of the chapter, when switching from the Django dev server to Gunicorn, and that was why we introduced WhiteNoise. Similarly, when we switch `DEBUG` off, WhiteNoise stops automagically finding static files in our code, and instead we need to run `collectstatic`:

Dockerfile (ch10l009)

```
WORKDIR /src

RUN python manage.py collectstatic

ENV DJANGO_DEBUG_FALSE=1

CMD ["gunicorn", "--bind", ":8888", "superlists.wsgi:application"]
```

Well, it was fiddly, but that should get us to passing tests after we build and run the Docker container!

```
$ docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -it superlists
```

And…

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]
OK
```

We're nearly ready to ship to production!

Let's quickly adjust our `gitignore`, as the static folder is in a new place, and do another commit to mark this bit of incremental progress:

```
$ git status
# should show dockerfile and untracked src/static folder
$ echo src/static >> .gitignore
$ git status
# should now be clean
$ git commit -am "Add collectstatic to dockerfile, and new location to gitignore"
```

Switching to a Nonroot User

Let's do one more! By default, Docker containers run as root. Although container security is a very well-tested ground by now, experts agree it's still good practice to use an unprivileged user inside your container.

The main fiddly thing, for us, will be dealing with permissions for the *db.sqlite3* file. It will need to be:

1. Writable by the nonroot user
2. In a *directory* that's writable by the nonroot user[8]

Making the Database Filepath Configurable

First, let's make the path to the database file configurable using an environment variable:

src/superlists/settings.py (ch10l011)

```
# SECURITY WARNING: don't run with debug turned on in production!
if "DJANGO_DEBUG_FALSE" in os.environ:
    DEBUG = False
    SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]
    ALLOWED_HOSTS = [os.environ["DJANGO_ALLOWED_HOST"]]
    db_path = os.environ["DJANGO_DB_PATH"]  ❶
else:
    DEBUG = True
    SECRET_KEY = "insecure-key-for-dev"
    ALLOWED_HOSTS = []
    db_path = BASE_DIR / "db.sqlite3"  ❷
[...]

# Database
# https://docs.djangoproject.com/en/5.2/ref/settings/#databases

DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": db_path  ❸
    }
}
```

8 This is surprising. It's due to SQLite wanting to write various additional temporary files during operation.

❶ Inside Docker, we'll assume that an environment variable called `DJANGO_DB_PATH` has been set. We save it to a local variable called `db_path`.

❷ Outside Docker, we'll use the default path to the database file.

❸ And we modify the `DATABASES` entry to use our `db_path` variable.

Now let's change the Dockerfile to set that env var, and to create and switch to our nonroot user, which we may as well call "nonroot" (although it could be anything!):

Dockerfile (ch10l012)

```
WORKDIR /src

RUN python manage.py collectstatic

ENV DJANGO_DEBUG_FALSE=1

RUN adduser --uid 1234 nonroot  ❶
USER nonroot  ❷

CMD ["gunicorn", "--bind", ":8888", "superlists.wsgi:application"]
```

❶ We use the `adduser` command to create our user, explicitly setting its UID to `1234`.[9]

❷ The `USER` directive in the Dockerfile tells Docker to run everything as that user by default.

Using UIDs to Set Permissions Across Host/Container Mounts

Our user will now have a writable home directory at `/home/nonroot`, so we'll put the database file in there. That takes care of the "writable directory" requirement.

Because we're mounting the file from outside though, that's not quite enough to make the file itself writable. We'll need to set the *owner* of the file to be `nonroot` as well. Because of the way Linux permissions work, we're going to use integer user IDs

9 A more or less arbitrary number, the first non-system user on a system is usually 1000, so it's nice that this won't be the same as the `elspeth` user outside the container. But other than that it could be any number greater than 1000 really.

(UIDs). This might seem a bit magical if you're not used to Linux permissions, so you'll have to trust me, I'm afraid.[10]

First, let's create a file with the right permissions, outside the container:

```
$ touch container.db.sqlite3

# Change the owner to uid 1234
$ sudo chown 1234 container.db.sqlite3

# This next step is needed on non-Linux dev environments,
# to make sure that the container host VM can write to the file.
# Change the file to be group-writeable as well as owner-writeable:
$ sudo chmod g+rw container.db.sqlite3
```

Now let's rebuild and run our container, changing the `--mount` path to our new file, and setting the `DJANGO_DB_PATH` environment variable to match:

```
$ docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/container.db.sqlite3",target=/home/nonroot/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -e DJANGO_DB_PATH=/home/nonroot/db.sqlite3 \
    -it superlists
```

As a first check that we can write to the database from inside the container, let's use `docker exec` to populate the database tables using `manage.py migrate`:

```
$ docker ps  # note container id
$ docker exec container-id-or-name python manage.py migrate
Operations to perform:
  Apply all migrations: auth, contenttypes, lists, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  [...]
  Applying lists.0001_initial... OK
  Applying lists.0002_item_text... OK
  Applying lists.0003_list... OK
  Applying lists.0004_item_list... OK
  Applying sessions.0001_initial... OK
```

10 Linux permissions aren't actually implemented using the string names of users; instead they use integer user IDs (called UIDs). The way we map from the UIDs to strings is using a special file called */etc/passwd*. Because */etc/passwd* is not the same inside and outside the container, the UIDs to username mappings inside and outside are not necessarily the same. However, the permission UIDs are just numbers, and they actually are stored inside individual files, so they don't change when you mount files. There's more info here on this Stack Overflow post.

And, as after every incremental change, we rerun our FT suite to make sure everything works:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]
OK
```

Great! We wrap up with a bit of housekeeping; we'll add this new database file to our `.gitignore`, and commit:

```
$ echo container.db.sqlite3 >> .gitignore
$ git commit -am"Switch to nonroot user"
```

Configuring Logging

One last thing we'll want to do is make sure that we can get logs out of our server. If things go wrong, we want to be able to get to the tracebacks. And as we'll soon see, switching `DEBUG` off means that Django's default logging configuration changes.

Provoking a Deliberate Error

To test this, we'll provoke a deliberate error by corrupting the database file:

```
$ echo bla > container.db.sqlite3
```

Now if you run the tests, you'll see they fail:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests --failfast
[...]

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
```

And you might spot in the browser that we just see a minimal error page, with no debug info, as in Figure 10-4 (try it manually if you like).

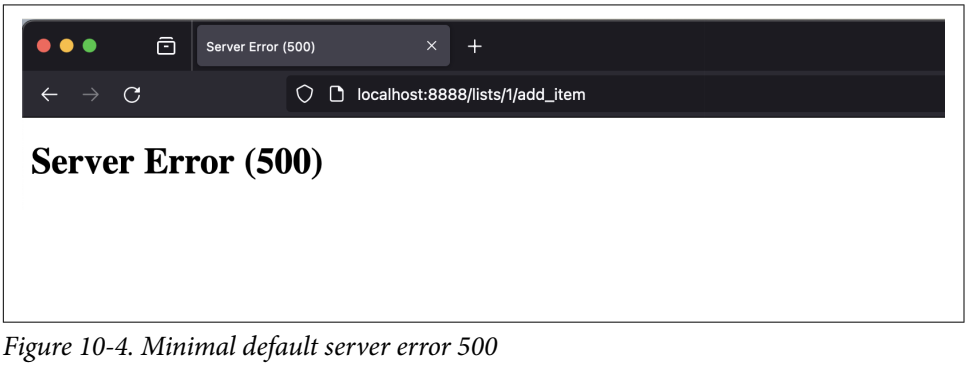


Figure 10-4. Minimal default server error 500

But if you look in your Docker terminal, you'll see there is no traceback:

```
[2024-02-28 10:41:53 +0000] [7] [INFO] Starting gunicorn 21.2.0
[2024-02-28 10:41:53 +0000] [7] [INFO] Listening at: http://0.0.0.0:8888 (7)
[2024-02-28 10:41:53 +0000] [7] [INFO] Using worker: sync
[2024-02-28 10:41:53 +0000] [8] [INFO] Booting worker with pid: 8
```

Where have the tracebacks gone? You might have been expecting that the Django debug page and its tracebacks would disappear from our web browser, but it's more of shock to see that they are no longer appearing in the terminal either! If you're like me, you might find yourself wondering if we really *did* see them earlier and starting to doubt your own sanity. But the explanation is that Django's default logging configuration changes when `DEBUG` is turned off.

This means we need to interact with the standard library's `logging` module, unfortunately one of the most fiddly parts of the Python standard library.[11]

Here's pretty much the simplest possible logging config, which just prints everything to the console (i.e., standard out); I've added this code to the very end of the *settings.py* file:

src/superlists/settings.py (ch10l013)

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {"class": "logging.StreamHandler"},
    },
    "loggers": {
        "root": {"handlers": ["console"], "level": "INFO"},
    },
}
```

Rebuild and restart our container…

```
$ docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -e DJANGO_DB_PATH=/home/nonroot/db.sqlite3 \
    -it superlists
```

11 It's not necessarily for bad reasons, but it is all very Java-ey and enterprise-y. I mean, yes, separating the concepts of handlers and loggers and filters, and making it all configurable in a nested hierarchy, is all well and good and covers every possible use case, but sometimes you just wanna say "just print stuff to stdout pls", and you wish that configuring the simplest thing was a little easier.

Then try the FT again (or submitting a new list item manually) and we now should see a clear error message:

```
Internal Server Error: /lists/new
Traceback (most recent call last):
[...]
  File "/src/lists/views.py", line 10, in new_list
    nulist = List.objects.create()
             ^^^^^^^^^^^^^^^^^^^^
[...]
  File "/venv/lib/python3.14/site-packages/django/db/backends/sqlite3/base.py",
  line 328, in execute
    return super().execute(query, params)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
django.db.utils.DatabaseError: file is not a database
```

We can fix and re-create the database by doing:

```
$ echo > container.db.sqlite3
$ docker exec -it <container_id> python manage.py migrate
```

And rerun the FTs to check we're back to a working state.

Let's do a final commit for this change:

```
$ git commit -am "Add logging config to settings.py"
```

Exercise for the Reader: Using the Django check Command

I don't have time in this book to cover every last aspect of production-readiness. Apart from anything else, this is a fast-changing area, and security updates to Django and its best practice recommandations change frequently, so things I write now might be incomplete by the time you read the book.

I *have* given a decent overview of the various different axes along which you'll need to make production-readiness changes, so hopefully you have a toolkit for how to do this sort of work.

If you'd like to dig into this a little bit more, or if you're preparing a real project for release into the wild, the next step is to read up on Django's deployment checklist.

The first suggestion is to use Django's "self-check" command, `manage.py check --deploy`. Here's what it reported as outstanding when I ran it in April 2025:

```
$ docker exec <container-id> python manage.py check --deploy
System check identified some issues:

WARNINGS:
?: (security.W004) You have not set a value for the SECURE_HSTS_SECONDS
setting. If your entire site is served only over SSL, you may want to consider
setting a value and enabling HTTP Strict Transport Security. Be sure to read
the documentation first; enabling HSTS carelessly can cause serious,
irreversible problems.
?: (security.W008) Your SECURE_SSL_REDIRECT setting is not set to True. Unless
your site should be available over both SSL and non-SSL connections, you may
want to either set this setting True or configure a load balancer or
reverse-proxy server to redirect all connections to HTTPS.
?: (security.W009) Your SECRET_KEY has less than 50 characters, less than 5
unique characters, or it's prefixed with django-insecure- indicating that it
was generated automatically by Django. Please generate a long and random value,
otherwise many of Django's security-critical features will be vulnerable to
attack.
?: (security.W012) SESSION_COOKIE_SECURE is not set to True. Using a
secure-only session cookie makes it more difficult for network traffic sniffers
to hijack user sessions.
?: (security.W016) You have django.middleware.csrf.CsrfViewMiddleware in your
MIDDLEWARE, but you have not set CSRF_COOKIE_SECURE to True. Using a
secure-only CSRF cookie makes it more difficult for network traffic sniffers to
steal the CSRF token.
```

Why not pick one of these and have a go at fixing it?

Wrap-Up

We might not have addressed every last issue that `check --deploy` raised, but we've at least touched on many or most of the things you might need to think about when considering production-readiness. We've worked in small steps and used our tests all the way along, and we're now ready to deploy our container to a real server!

Find out how, in our next exciting installment…

One more recommendation for PythonSpeed and its Docker Packaging for Python Developers article—again, I cannot recommend it highly enough. Read it before you're too much older!

Production-Readiness Config

A few things to think about when trying to prepare a production-ready configuration:

Don't use the Django dev server in production
: Something like Gunicorn or uWSGI is a better tool for running Django; it will let you run multiple workers, for example.

Decide how to serve your static files
: Static files aren't the same kind of things as the dynamic content that comes from Django and your web app, so they need to be treated differently. WhiteNoise is just one example of how you might do that.

Check your settings.py for dev-only config
: `DEBUG=True`, `ALLOWED_HOSTS`, and `SECRET_KEY` are the ones we came across, but you will probably have others (and we'll see more when we start to send emails from the server).

Change things one at a time and rerun your tests frequently
: Whenever we make a change to our server configuration, we can rerun the test suite, and either be confident that everything works as well as it did before, or find out immediately if we did something wrong.

Think about logging and observability
: When things go wrong, you need to be able to find out what happened. At a minimum, you need a way of getting logs and tracebacks out of your server, and in more advanced environments you'll want to think about metrics and tracing too. But we can't cover all that in this book!

Use the Django "check" command
: `python manage.py check --deploy` can give you a list of additional settings to check for production-readiness.

CHAPTER 11

Getting a Server Ready for Deployment

This chapter is all about getting ready for our deployment. We're going to spin up an actual server, make it accessible on the internet with a real domain name, and set up the authentication and credentials we need to be able to control it remotely with SSH and Ansible.

Manually Provisioning a Server to Host Our Site

We can separate our "deployment" into two tasks:

1. *Provisioning* a new server to be able to host the code, which includes choosing an operating system, getting basic credentials to log in, and configuring DNS
2. *Deploying* our application to an existing server, which includes getting our Docker image onto the server, starting a container, and configuring it to talk to the database and the outside world

Infrastructure-as-code tools can let you automate both of these, but the provisioning parts tend to be quite vendor-specific, so for the purposes of this book, we can live with manual provisioning.

I should probably stress once more that deployment is something that varies a lot and, as a result, there are few universal best practices for how to do it. So, rather than trying to remember the specifics of what I'm doing here, you should be trying to understand the rationale, so that you can apply the same kind of thinking in the specific future circumstances you encounter.

Choosing Where to Host Our Site

There are loads of different solutions out there these days, but they broadly fall into two camps:

1. Running your own (probably virtual) server—aka VPS (virtual private server)
2. Using a platform as a service (PaaS) offering like Heroku or my old employers, PythonAnywhere

With a PaaS, you don't get your own server; instead, you're renting a "service" at a higher level of abstraction. Particularly for small sites, a PaaS offers many advantages over running your own server, and I would definitely recommend looking into them. We're not going to use a PaaS in this book, however, for several reasons. The main reason is that I want to avoid endorsing specific commercial providers. Secondly, all the PaaS offerings are quite different, and the procedures to deploy to each vary a lot—learning about one doesn't necessarily tell you about the others. Any one of them might radically change their process or business model by the time you get to read this book.

Instead, we'll learn just a tiny bit of good old-fashioned server admin, including SSH and manual debugging. They're unlikely to ever go away, and knowing a bit about them will get you some respect from all the grizzled dinosaurs out there.

Spinning Up Our Own Server

I'm not going to dictate how you spin up a server—whether you choose Amazon AWS, Rackspace, DigitalOcean, your own server in a datacentre, or a Raspberry Pi in a cupboard under the stairs, any solution should be fine, as long as:

- Your server is running Ubuntu 22.04 (aka "Jammy/LTS").
- You have root access to it.
- It's on the public internet (i.e., it has a public IP address).
- You can SSH into it (I recommend using a nonroot user account, with `sudo` access, and public/private key authentication).

I'm recommending Ubuntu as a distro because it's popular and I'm used to it.[1] If you know what you're doing, you can probably get away with using something else, but I won't be able to help you as much if you get stuck.

1 Linux as an operating system comes in lots of different flavours, called "distros" or "distributions". The differences between them and their relative pros and cons are, like any seemingly minor detail, of tremendous interest to the right kind of nerd. We don't need to care about them for this book. As I say, Ubuntu is fine.

Step-by-Step Instructions for Spinning Up a Server

I appreciate that, if you've never started a Linux server before and you have absolutely no idea where to start, this is a big ask, especially when I'm refusing to "dictate" exactly how to do it.

With that in mind, I wrote a very brief guide on GitHub. I didn't want to include it in the book itself because, inevitably, I do end up specifying a specific commercial provider in there.

Some people get to this chapter, and are tempted to skip the domain bit and the "getting a real server" bit, and just use a VM on their own PC. Don't do this. It's *not* the same, and you'll have more difficulty following the instructions, which are complicated enough as it is. If you're worried about cost, have a look at the guide I wrote for free options.

General Tip for Working with Infrastructure

The most important lesson to remember over the next few chapters is, as always but more than ever, to work incrementally, make one change at a time, and run your tests frequently.

When things (inevitably) go wrong, resist the temptation to flail about and make other unrelated changes in the hope that things will start working again; instead, stop, go backwards if necessary to get to a working state, and figure out what went wrong before moving forwards again.

It's just as easy to fall into the Refactoring Cat trap when working with infrastructure!

Getting a Domain Name

We're going to need a couple of domain names at this point in the book—they can both be subdomains of a single domain. I'm going to use *superlists.ottg.co.uk* and *staging.ottg.co.uk*. If you don't already own a domain, this is the time to register one! Again, this is something I really want you to *actually* do. If you've never registered a domain before, just pick any old registrar and buy a cheap one—it should only cost you $5 or so, and I promise seeing your site on a "real" website will be a thrill.

Configuring DNS for Staging and Live Domains

We don't want to be messing about with IP addresses all the time, so we should point our staging and live domains to the server. At my registrar, the control screens looked a bit like Figure 11-1.

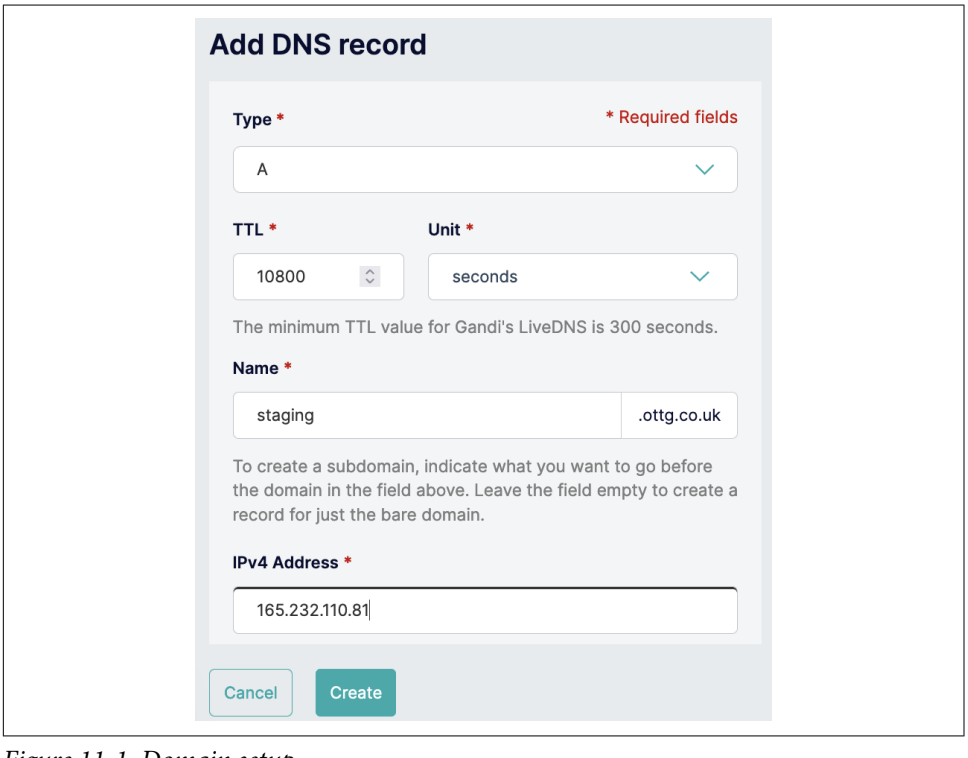


Figure 11-1. Domain setup

In the DNS system, pointing a domain at a specific IP address is referred to as an "A-record".[2] All registrars are slightly different, but a bit of clicking around should get you to the right screen in yours. You'll need two A-records: one for the staging address and one for the live one. No need to worry about any other type of record.

DNS records take some time to "propagate" around the world (it's controlled by a setting called "TTL", time to live), so once you've set up your A-record, you can check its progress on a "propagation checking" service like this one: *https://www.whatsmydns.net/#A/staging.ottg.co.uk*.

2 Strictly speaking, A-records are for IPv4, and you can also use AAAA-records for IPv6. Some cheap providers only support IPv6, and there's nothing wrong with that.

I'm planning to host my staging server at *staging.ottg.co.uk*.

Ansible

Infrastructure-as-code tools, also called "configuration management" tools, come in lots of shapes and sizes. Chef and Puppet were two of the original ones, and you'll probably come across Terraform, which is particularly strong on managing cloud services like AWS.

We're going to use the infrastructure automation tool Ansible—because it's relatively popular, because it can do everything we need it to, because I'm biased that it happens to be written in Python, and because it's probably the one I'm personally most familiar with.

Another tool could probably have worked just as well! The main thing to remember is the *concept*, which is that, as much as possible we want to manage our server configuration *declaratively*, by expressing the desired state of the server in a particular configuration syntax, rather than specifying a procedural series of steps to be followed one by one.

Ansible Versus SSH: How We'll Talk to Our Server

Figure 11-2 shows how we'll interact with our server using SSH, Ansible, and our FTs.

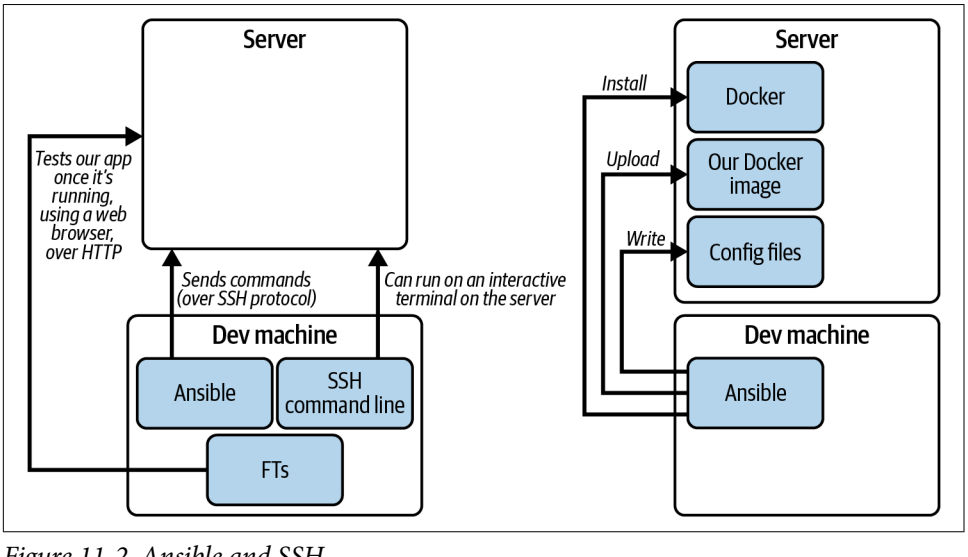


Figure 11-2. Ansible and SSH

Our objective is to use Ansible to automate the process of deploying to our server: making sure that the server has everything it needs to run our app (mostly, Docker and our container image), and then telling it to start or restart our container.

Now and again, we'll want to "log on" to the server and have a look around manually; for that, we'll use the `ssh` command line on our computer, which can let us open up an interactive console on the server.

Finally, we'll run our FTs against the server, once it's running our app, to make sure it's all working correctly.

Start by Making Sure We Can SSH In

At this point and for the rest of the book, I'm assuming that you have a nonroot user account set up, and that it has `sudo` privileges, so whenever we need to do something that requires root access, we use `sudo`, (or "become" in Ansible terminology); I'll be explicit about that in the various instructions that follow.

My user is called "elspeth", but you can call yours whatever you like! Just remember to substitute it in all the places I've hardcoded it. See the guide I wrote ("Step-by-Step Instructions for Spinning Up a Server" on page 247) if you need tips on creating a `sudo` user.

Ansible uses SSH under the hood to talk to the server, so checking we can log in "manually" is a good first step:

```
$ ssh elspeth@staging.ottg.co.uk
elspeth@server$: echo "hello world"
hello world
```

Look out for that `elspeth@server` in the command-line listings in this chapter. It indicates commands that must be run on the server, as opposed to commands you run on your own PC.

Use WSL on Windows

Ansible will not run natively on Windows (see the docs) but you can use the Windows Subsystem for Linux (WSL), a sort of mini-Linux that Microsoft has made to run inside Windows.

Follow Microsoft's instructions for setting up WSL.

Once inside your WSL environment, you can navigate to your project directory on the host Windows filesystem at */mnt/c/Users/yourusername/Projects/superlists*, for example.

You'll need to use a different virtualenv for WSL:

```
yourusername@wsl: cd /mnt/c/Users/yourusername/Projects/superlists
yourusername@wsl: python -m venv .venv-wsl
yourusername@wsl: source .venv-wsl/bin/activate
```

If you are using public key authentication, it's probably simplest to generate a new SSH keypair, and add it to *home/elspeth/.ssh/authorized_keys* on the server:

```
yourusername@wsl: ssh-keygen
[..]
yourusername@wsl: cat ~/.ssh/.pub*
# copy the public key to your clipboard,
```

I'd suggest you *only* use WSL when you need to use Ansible.

The alternative is to switch your whole dev environment to WSL, and move your source code in there, but you might need to overcome a few hurdles around things like networking.

Debugging Issues with SSH

Here's a few things to try if you can't SSH in.

Debugging network connectivity

First, check network connectivity: can we even reach the server?

```
$ ping staging.ottg.co.uk

# if that doesn't work, try the IP address
$ ping 193.184.215.14  # or whatever your IP is

# also see if the domain name resolves
$ nslookup staging.ottg.co.uk
```

If the IP works and the domain name doesn't, and/or if the `nslookup` doesn't work, you should go check your DNS config at your registrar. You may just need to wait! Try a DNS propagation checker like *https://www.whatsmydns.net/#A/staging.ottg.co.uk*.

Debugging SSH auth issues

Next, let's try and debug any possible issues with authentication.

First, your hosting provider might have the option to open a console directly from within their web UI. That's worth trying, and if there are any problems there, then you probably need to restart your server, or perhaps stop it and create a new one.

It's worth double-checking your IP address at this point, in your provider's server control panel pages.

Next, we can try debugging our SSH connection:

```
# try the -v flag which turn on verbose/debug output
$ ssh -v elspeth@staging.ottg.uk
OpenSSH_9.7p1, LibreSSL 3.3.6
debug1: Reading configuration data ~/.ssh/config
debug1: Reading configuration data ~/.colima/ssh_config
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: /etc/ssh/ssh_config line 21: include /etc/ssh/ssh_config.d/* matched no files
debug1: /etc/ssh/ssh_config line 54: Applying options for *
debug1: Authenticator provider $SSH_SK_PROVIDER did not resolve; disabling
debug1: Connecting to staging.ottg.uk port 22.
ssh: Could not resolve hostname staging.ottg.uk: nodename nor servname provided, or not
known
# oops I made a typo!  it should be ottg.co.uk not ottg.uk
```

If that doesn't help, try switching to the root user instead:

```
$ ssh -v root@staging.ottg.co.uk
[...]
debug1: Authentications that can continue: publickey
debug1: Next authentication method: publickey
debug1: get_agent_identities: bound agent to hostkey
debug1: get_agent_identities: agent returned 1 keys
debug1: Will attempt key: ~/.ssh/id_ed25519 ED25519 SHA256:gZLxb9zCuGVT1Dm8 [...]
debug1: Will attempt key: ~/.ssh/id_rsa
debug1: Will attempt key: ~/.ssh/id_ecdsa
debug1: Will attempt key: ~/.ssh/id_ecdsa_sk
debug1: Will attempt key: ~/.ssh/id_ed25519_sk
debug1: Will attempt key: ~/.ssh/id_xmss
debug1: Will attempt key: ~/.ssh/id_dsa
debug1: Offering public key: ~/.ssh/id_ed25519 [...]
debug1: Server accepts key: ~/.ssh/id_ed25519 [...]
Authenticated to staging.ottg.co.uk ([165.232.110.81]:22) using "publickey".
```

That one actually worked! But in the verbose output, you can watch to make sure it finds the right SSH keys, for example.

If root works but your nonroot user doesn't, you may need to add your public key to `/home/yournonrootuser/.ssh/authorized_keys`.

If root doesn't work either, you may need to add your public SSH key to your account settings page, via your provider's web UI. That may or may not take effect immediately; you might need to delete your old server and create a new one.

Remember, that probably means a new IP address!

Security

A serious discussion of server security is beyond the scope of this book, and I'd warn against running your own servers without learning a good bit more about it. (One reason people choose to use a PaaS to host their code is that it means slightly fewer security issues to worry about.) If you'd like a place to start, here's as good a place as any: *https://blog.codelitt.com/my-first-10-minutes-on-a-server-primer-for-securing-ubuntu*.

I can definitely recommend the eye-opening experience of installing Fail2Ban and watching its logfiles to see just how quickly it picks up on random drive-by attempts to brute force your SSH login. The internet is a wild place!

Installing Ansible

Assuming we can reliably SSH into the server, it's time to install Ansible and make sure it can talk to our server as well.

Take a look at the Ansible installation guide for all the various options, but probably the simplest thing to do is to install Ansible into the virtualenv on our local machine (Ansible doesn't need to be installed on the server):

```
$ pip install ansible
# we also need the Docker SDK for the ansible/docker integration to work:
$ pip install docker
```

Checking Ansible Can Talk to Our Server

This is the last step in ensuring we're ready: making sure Ansible can talk to our server.

At the core of Ansible is what's called a "playbook", which describes what we want to happen on our server. Let's create one now. It's probably a good idea to keep it in a folder of its own:

```
mkdir infra
```

And here's a minimal playbook whose job is just to "ping" the server, to check we can talk to it. It's in a format called YAML (yet another markup language) which, if you've never come across before, you will soon develop a love-hate relationship for:[3]

infra/deploy-playbook.yaml (ch11l001)

```
- hosts: all
  tasks:
    - name: Ping to make sure we can talk to our server
      ansible.builtin.ping:
```

We won't worry too much about the syntax or how it works at the moment; let's just use it to make sure everything works.

To invoke Ansible, we use the command `ansible-playbook`, which will have been installed into your virutalenv when we did the `pip install ansible` earlier.

Here's the full command we'll use, with an explanation of each part:

```
ansible-playbook \
  --user=elspeth \ ❶
  -i staging.ottg.co.uk, \ ❷❸
  infra/deploy-playbook.yaml \ ❹
  -vv ❺
```

❶ The `--user=` flag lets us specify the user to use to authenticate with the server. This should be the same user you can SSH with.

❷ The `-i` flag specifies what server to run against.

❸ Note the trailing comma after the server hostname. Without this, it won't work (it's there because Ansible is designed to work against multiple servers at the same time).[4]

3 The "love" part is that YAML is very easy to *read* and scan through at a glance. The "hate" part is that the actual syntax is surprisingly fiddly to get right: the difference between lists and key/value maps is subtle and I can never quite remember it, honestly.

❹ Next comes the path to our playbook, as a positional argument.

❺ Finally the `-v` or `-vv` flags control how verbose the output will be—useful for debugging!

Here's some example output when I run it:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -vv
ansible-playbook [core 2.17.5]
  config file = None
  configured module search path = ['~/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location =
...goat-book/.venv/lib/python3.14/site-packages/ansible
  ansible collection location =
~/.ansible/collections:/usr/share/ansible/collections
  executable location = ...goat-book/.venv/bin/ansible-playbook
  python version = 3.14.0 (main, Oct 11 2024, 22:59:05) [Clang 15.0.0
(clang-1500.3.9.4)] (...goat-book/.venv/bin/python)
  jinja version = 3.1.4
  libyaml = True
No config file found; using defaults
Skipping callback 'default', as we already have a stdout callback.
Skipping callback 'minimal', as we already have a stdout callback.
Skipping callback 'oneline', as we already have a stdout callback.

PLAYBOOK: deploy-playbook.yaml *************************************************
1 plays in infra/deploy-playbook.yaml

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
task path: ...goat-book/infra/deploy-playbook.yaml:1
[WARNING]: Platform linux on host staging.ottg.co.uk is using the discovered
Python interpreter at /usr/bin/python3.10, but future
installation of another Python interpreter could change the meaning of that
path. See https://docs.ansible.com/ansible-
core/2.17/reference_appendices/interpreter_discovery.html for more information.
ok: [staging.ottg.co.uk]

TASK [Ping to make sure we can talk to our server] *****************************
task path: ...goat-book/infra/deploy-playbook.yaml:3
ok: [staging.ottg.co.uk] => {"changed": false, "ping": "pong"}

PLAY RECAP *********************************************************************
staging.ottg.co.uk         : ok=2    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Looking good! In the next chapter, we'll use Ansible to get our app up and running on our server. It'll be a thrill, I promise!

4 The "i" in the `-i` flag stands for "inventory". Using the `-i` flag is actually a little unconventional. If you read the Ansible docs, you'll find they usually recommend having an "inventory file", which lists all your servers, along with various bits of qualifying metadata. That's overkill for our use case though!

Server Prep Recap

VPS versus PaaS
: We discussed the trade-offs of running your own server versus opting for a PaaS. A VPS is great for learning, but you might find the lower admin overhead of a PaaS makes sense for real projects.

Domain name registration and DNS
: This tends to be something you only do once, but buying a domain name and pointing it at your server is an unavoidable part of hosting a web app. Now you know your TTLs from your A-records!

SSH
: SSH is the Swiss Army knife of server admin. The dream is that everything is automated, but now and again you just gotta open up a shell on that box!

Ansible
: Ansible will be our deployment automation tool. We've had the barest of teasers, but we have it installed and we're ready to learn how to use it.

CHAPTER 12

Infrastructure as Code: Automated Deployments with Ansible

> Automate, automate, automate.
>
> —Cay S. Horstmann

Now that our server is up and running, we want to install our app on it, using our Docker image and container.

We *could* do this manually, but a key insight of modern software engineering is that small, frequent deployments are a must.

This insight about the importance of frequent deployments we owe to Nicole Forsgren and the *State of DevOps* reports. They are some of the only really firm science we have in the field of software engineering.

Frequent deployments rely on automation,[1] so we'll use Ansible.

1 Some readers mentioned a worry that using automation tools would leave them with less understanding of the underlying infrastructure. But in fact, using automation requires deep understanding of the things you're automating. So, don't worry; we'll be taking the time to look under the hood and make sure we know how things work.

Automation is also key to making sure our tests give us true confidence over our deployments. If we go to the trouble of building a staging server,[2] we want to make sure that it's as similar as possible to the production environment. By automating the way we deploy, and using the same automation for staging and prod, we give ourselves much more confidence.

The buzzword for automating your deployments these days is "infrastructure as code" (IaC).

Why not ping me a note once your site is live on the web, and send me the URL? It always gives me a warm and fuzzy feeling…Email me at *obeythetestinggoat@gmail.com*.

A First Cut of an Ansible Playbook for Deployment

Let's start using Ansible a little more seriously. We're not going to jump all the way to the end though! Baby steps, as always. Let's see if we can get it to run a simple "hello world" Docker container on our server.

2 Depending on where you work, what I'm calling a "staging" server, some people would call a "development" server, and some others would also like to distinguish "preproduction" servers. Whatever we call it, the point is to have somewhere we can try our code out in an environment that's as similar as possible to the real production server. As we'll see, Docker isn't *quite* enough!

Let's delete the old content, which had the "ping", and replace it with something like this:

infra/deploy-playbook.yaml (ch12l001)

```
---
- hosts: all

  tasks:

    - name: Install docker  ❶
      ansible.builtin.apt:  ❷
        name: docker.io  ❸
        state: latest
        update_cache: true
      become: true

    - name: Run test container
      community.docker.docker_container:
        name: testcontainer
        state: started
        image: busybox
        command: echo hello world
      become: true
```

❶ An Ansible playbook is a series of "tasks"; we now have more than one. In that sense, it's still quite sequential and procedural, but the individual tasks themselves are quite declarative. Each one usually has a human-readable `name` attribute.

❷ Each task uses an Ansible "module" to do its work. This one uses the `built in.apt` module, which provides a wrapper around the `apt` Debian and Ubuntu package management tool.

❸ Each module then provides a bunch of parameters that control how it works. Here, we specify the `name` of the package we want to install ("docker.io"[3]) and tell it to update its cache first, which is required on a fresh server.

Most Ansible modules have pretty good documentation—check out the `builtin.apt` one for example; I often skip to the "Examples" section.

3 In the official Docker installation instructions, you'll see a recommendation to install Docker via a private package repository. I wanted to avoid that complexity for the book, but you should probably follow those instructions in a real-world scenario, to make sure your version of Docker has all the latest security patches.

Let's rerun our deployment command, `ansible-playbook`, with the same flags we used in the last chapter:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -vv
ansible-playbook [core 2.16.3]
  config file = None
  [...]
No config file found; using defaults
BECOME password:
Skipping callback 'default', as we already have a stdout callback.
Skipping callback 'minimal', as we already have a stdout callback.
Skipping callback 'oneline', as we already have a stdout callback.

PLAYBOOK: deploy-playbook.yaml *************************************************
1 plays in infra/deploy-playbook.yaml

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:2
ok: [staging.ottg.co.uk]
PLAYBOOK: deploy-playbook.yaml *************************************************
1 plays in infra/deploy-playbook.yaml

TASK [Install docker] **********************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:6
ok: [staging.ottg.co.uk] => {"cache_update_time": 1708981325, "cache_updated":
true, "changed": false}


TASK [Install docker] **********************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:6
changed: [staging.ottg.co.uk] => {"cache_update_time": [...]
"cache_updated": true, "changed": true, "stderr": "", "stderr_lines": [],
"stdout": "Reading package lists...\nBuilding dependency tree...\nReading [...]
information...\nThe following additional packages will be installed:\n
wmdocker\nThe following NEW packages will be installed:\n  docker wmdocker\n0

TASK [Run test container] ******************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:13
changed: [staging.ottg.co.uk] => {"changed": true, "container":
{"AppArmorProfile": "docker-default", "Args": ["hello", "world"], "Config":
[...]

PLAY RECAP *********************************************************************
staging.ottg.co.uk         : ok=3    changed=2    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

I don't know about you, but whenever I make a terminal spew out a stream of output, I like to make little *brrp brrp brrp* noises—a bit like the computer, Mother, in *Alien*. Ansible scripts are particularly satisfying in this regard.

You may need to use the `--ask-become-pass` argument to `ansible-playbook` if you get an error, "Missing sudo password".[4]

Idempotence and Declarative Configuration

IaC tools like Ansible aim to be "declarative", meaning that, as much as possible, you specify the desired state that you want, rather than specifying a series of steps to get there.

This concept goes along with the idea of "idempotence", which is when you want a thing that has the same effect, whether it is run just once or multiple times.

An example is the `apt` module that we used to install Docker. It doesn't crash if Docker is already installed and, in fact, Ansible is smart enough to check first before trying to install anything. It makes no difference whether you run it once or many times.

In contrast, adding an item to our to-do list is not currently idempotent. If I add "Buy milk" and then I add "Buy milk" again, I end up with two items that both say "Buy milk". (We might fix that later, mind you.)

SSHing Into the Server and Viewing Container Logs

Ansible *looks* like it's doing its job, but let's practice our SSH skills, and do some good old-fashioned system admin. Let's log in to our server and see if we can see any actual evidence that our container has run.

After we `ssh` in, we can use `docker ps`, just like we do on our own machine. We pass the `-a` flag to view *all* containers, including old/stopped ones. Then we can use `docker logs` to view the output from one of them:

4 You can also look into "passwordless sudo" if it's all just too annoying, but that does have security implications.

```
$ ssh elspeth@staging.superlists.ottg.co.uk
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 5.15.0-67-generic x86_64)
 [...]

elspeth@server$ sudo docker ps -a
CONTAINER ID   IMAGE     COMMAND              CREATED      STATUS
PORTS     NAMES
3a2e600fbe77   busybox   "echo hello world"   2 days ago   Exited (0) 10
minutes ago             testcontainer

elspeth@server:$ sudo docker logs testcontainer
hello world
```

Look out for that `elspeth@server` in the command-line listings in this chapter. It indicates commands that must be run on the server, as opposed to commands you run on your own PC.

SSHing in to check things worked is a key server debugging skill! It's something we want to practice on our staging server, because ideally we'll want to avoid doing it on production machines.

Docker Debugging

Here's a rundown of some of the debugging tools—some we've already seen and some new ones we'll use in this chapter. When things don't go to plan, they can help shed some light. All of them should be run on the server, inside an SSH session:

- You can check the Container logs using `docker logs superlists`.
- You can run things "inside" the container with `docker exec <container-id-or-name> <cmd>`. A couple of useful examples include `docker exec superlists env`, to print environment variables, and just `docker exec -it superlists bash` to open an interactive Bash shell, inside the container.
- You can get lots of detailed info on the *container* using `docker inspect superlists`. This is a good place to go check on environment variables, port mappings, and exactly which image was running, for example.
- You can get detailed info on the *image* with `docker image inspect superlists`. You might need this to check the exact image hash, to make sure it's the same one you built locally.

Allowing Rootless Docker Access

Having to use `sudo` or `become=True` to run Docker commands is a bit of a pain. If we add our user to the `docker` group, we can run Docker commands without `sudo`:

infra/deploy-playbook.yaml (ch12l001-1)

```
- name: Install docker
      [...]

- name: Add our user to the docker group, so we don't need sudo/become
  ansible.builtin.user:  ❶
    name: '{{ ansible_user }}'  ❷
    groups: docker
    append: true  # don't remove any existing groups.
  become: true

- name: Reset ssh connection to allow the user/group change to take effect
  ansible.builtin.meta: reset_connection  ❸

- name: Run test container  ❹
      [...]
```

❶ We use the `builtin.user` module to add our user to the `docker` group.

❷ The `{{ ... }}` syntax enables us to interpolate some variables into our config file, much like in a Django template. `ansible_user` will be the user we're using to connect to the server—i.e., "elspeth", in my case.

❸ As per the task name, we need this for the user/group change to take effect. Strictly speaking, this is only needed the first time we run the script; if you've got some time, you can read up on how to make tasks conditional and configure it to only run if the `builtin.user` tasks has actually made a change.

❹ We can remove the `become: true` from this task and it should still work.

Let's run that:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -vv
PLAYBOOK: deploy-playbook.yaml *************************************************
1 plays in infra/deploy-playbook.yaml

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
[...]
ok: [staging.ottg.co.uk]

TASK [Install docker] **********************************************************
[...]
ok: [staging.ottg.co.uk] => {"cache_update_time": 1738767216, "cache_updated":
true, "changed": false}

TASK [Add our user to the docker group, so we don't need sudo/become] *********
[...]
changed: [staging.ottg.co.uk] => {"append": false, "changed": true, [...]
"", "group": 1000, "groups": "docker", [...]

TASK [Reset ssh connection to allow the user/group change to take effect] *****
[...]
META: reset connection

TASK [Run test container] ******************************************************
[...]
changed: [staging.ottg.co.uk] => {"changed": true, "container": [...]

PLAY RECAP *********************************************************************
staging.ottg.co.uk         : ok=4    changed=2    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

And check that it worked:

```
elspeth@server$ docker ps -a  # no sudo yay!
CONTAINER ID   IMAGE        COMMAND                  CREATED          STATUS
PORTS     NAMES
bd3114e43f55   busybox      "echo hello world"       12 minutes ago   Exited (0)
6 seconds ago               testcontainer

elspeth@server$ docker logs testcontainer
hello world
hello world
```

Sure enough, we no longer need `sudo`, and we can see that a new version of the container just ran.

You know, that's worthy of a commit!

```
$ git add infra/deploy-playbook.yaml
$ git commit -m "Made a start on an ansible playbook for deployment"
```

Let's move on to trying to get our actual Docker container running on the server. As we go through, you'll see that we're going to work through very similar issues to the ones we've already figured our way through in the last couple of chapters:

- Configuration
- Networking
- The database

Getting Our Image Onto the Server

Typically, you can "push" and "pull" container images to a "container registry"—Docker offers a public one called Docker Hub, and organisations will often run private ones, hosted by cloud providers like AWS.

So your process of getting an image onto a server is usually:

1. Push the image from your machine to the registry.
2. Pull the image from the registry onto the server. Usually this step is implicit, in that you just specify the image name in the format `registry-url/image-name:tag`, and then `docker run` takes care of pulling down the image for you.

But I don't want to ask you to create a Docker Hub account, nor implicitly endorse any particular provider, so we're going to "simulate" this process by doing it manually.

It turns out you can "export" a container image to an archive format, manually copy that to the server, and then reimport it. In Ansible config, it looks like this:

infra/deploy-playbook.yaml (ch12l002)

```
- name: Install docker
    [...]
- name: Add our user to the docker group, so we don't need sudo/become
    [...]
- name: Reset ssh connection to allow the user/group change to take effect
    [...]

- name: Export container image locally  ❶
  community.docker.docker_image:
    name: superlists
    archive_path: /tmp/superlists-img.tar
    source: local
  delegate_to: 127.0.0.1

- name: Upload image to server  ❷
  ansible.builtin.copy:
    src: /tmp/superlists-img.tar
    dest: /tmp/superlists-img.tar

- name: Import container image on server  ❸
  community.docker.docker_image:
    name: superlists
    load_path: /tmp/superlists-img.tar
    source: load
    force_source: true  ❹
    state: present

- name: Run container
  community.docker.docker_container:
    name: superlists
    image: superlists  ❺
    state: started
    recreate: true  ❻
```

❶ We export the Docker image to a *.tar* file by using the `docker_image` module with the `archive_path` set to a tempfile, and setting the `delegate_to` attribute to say we're running that command on our local machine rather than the server.

❷ We then use the `copy` module to upload the *.tar* file to the server.

❸ And we use `docker_image` again, but this time with `load_path` and `source: load` to import the image back on the server.

❹ The `force_source` flag tells the server to attempt the import, even if an image of that name already exists.

❺ We change our "run container" task to use the `superlists` image, and we'll use that as the container name too.

❻ Similarly to `source: load`, the `recreate` argument tells Ansible to re-create the container even if there's already one running whose name and image match "superlists".

If you see an error saying "Error connecting: Error while fetching server API version", it may be because the Python Docker software development kit (SDK) can't find your Docker daemon. Try restarting Docker Desktop if you're on Windows or a Mac. If you're not using the standard Docker engine—with Colima or Podman, for example—you may need to set the `DOCKER_HOST` environment variable (e.g., `DOCKER_HOST=unix:///$HOME/.colima/default/docker.sock`) or use a symlink to point to the right place. See the Colima FAQ or Podman docs.

Let's run the new version of our playbook, and see if we can upload a Docker image to our server and get it running:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -vv
[...]

PLAYBOOK: deploy-playbook.yaml *************************************************
1 plays in infra/deploy-playbook.yaml

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:2
ok: [staging.ottg.co.uk]

TASK [Install docker] **********************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:5
ok: [staging.ottg.co.uk] => {"cache_update_time": 1708982855, "cache_updated":
false, "changed": false}
TASK [Add our user to the docker group, so we don't need sudo/become] *********
task path: ...goat-book/infra/deploy-playbook.yaml:11
ok: [staging.ottg.co.uk] => {"append": false, "changed": false, [...]

TASK [Reset ssh connection to allow the user/group change to take effect] *****
task path: ...goat-book/infra/deploy-playbook.yaml:17
META: reset connection

TASK [Export container image locally] ******************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:20
changed: [staging.ottg.co.uk -> 127.0.0.1] => {"actions": ["Archived image
superlists:latest to /tmp/superlists-img.tar, overwriting archive with image
11ff3b83873f0fea93f8ed01bb4bf8b3a02afa15637ce45d71eca1fe98beab34 named
superlists:latest"], "changed": true, "image": {"Architecture": "amd64",
[...]

TASK [Upload image to server] **************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:27
changed: [staging.ottg.co.uk] => {"changed": true, "checksum":
"313602fc0c056c9255eec52e38283522745b612c", "dest": "/tmp/superlists-img.tar",
[...]

TASK [Import container image on server] ****************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:32
changed: [staging.ottg.co.uk] => {"actions": ["Loaded image superlists:latest
from /tmp/superlists-img.tar"], "changed": true, "image": {"Architecture":
"amd64", "Author": "", "Comment": "buildkit.dockerfile.v0", "Config":
[...]

TASK [Run container] ***********************************************************
task path: ...goat-book/superlists/infra/deploy-playbook.yaml:40
changed: [staging.ottg.co.uk] => {"changed": true, "container":
{"AppArmorProfile": "docker-default", "Args": ["--bind", ":8888",
"superlists.wsgi:application"], "Config": {"AttachStderr": true, "AttachStdin":
false, "AttachStdout": true, "Cmd": ["gunicorn", "--bind", ":8888",
"superlists.wsgi:application"], "Domainname": "", "Entrypoint": null, "Env":
[...]
staging.ottg.co.uk         : ok=7    changed=4    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

That looks good!

For completeness, let's also add a step to explicitly build the image locally (this means we aren't dependent on having run `docker build` locally):

infra/deploy-playbook.yaml (ch12l003)

```
- name: Reset ssh connection to allow the user/group change to take effect
  [...]

- name: Build container image locally
  community.docker.docker_image:
    name: superlists
    source: build
    state: present
    build:
      path: ..
      platform: linux/amd64  ❶
    force_source: true
  delegate_to: 127.0.0.1

- name: Export container image locally
  [...]
```

❶ I needed this `platform` attribute to work around an issue with compatibility between Apple's new ARM-based chips and our server's x86/AMD64 architecture. You could also use this `platform:` to cross-build Docker images for a Raspberry Pi from a regular PC, or vice versa. It does no harm in any case.

Taking a Look Around Manually

Time to take another proverbial look under the hood, to check whether it really worked. Hopefully we'll see a container that looks like ours:

```
$ ssh elspeth@staging.superlists.ottg.co.uk
Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 5.15.0-67-generic x86_64)
 [...]

elspeth@server$ docker ps -a
CONTAINER ID   IMAGE     COMMAND              CREATED      STATUS
PORTS     NAMES
3a2e600fbe77   busybox   "echo hello world"   2 days ago   Exited (0) 10
minutes ago             testcontainer
129e36a42190   superlists   "/bin/sh -c \'gunicor…"   About a minute ago
Exited (3) About a minute ago             superlists
```

OK! We can see our "superlists" container is there now, both named "superlists" and based on an image called "superlists".

The `Status: Exited` is a bit more worrying though.

Still, that's a good bit of progress, so let's do a commit (back on your own machine):

```
$ git commit -am"Build our image, use export/import to get it on the server, try and run it"
```

Docker logs

Now, back on the server, let's take a look at the logs of our new container to see if we can figure out what's happened:

```
elspeth@server:$ docker logs superlists
[2024-02-26 22:19:15 +0000] [1] [INFO] Starting gunicorn 21.2.0
[2024-02-26 22:19:15 +0000] [1] [INFO] Listening at: http://0.0.0.0:8888 (1)
[2024-02-26 22:19:15 +0000] [1] [INFO] Using worker: sync
[...]
  File "/src/superlists/settings.py", line 22, in <module>
    SECRET_KEY = os.environ["DJANGO_SECRET_KEY"]
                 ~~~~~~~^^^^^^^
  File "<frozen os>", line 685, in getitem
KeyError: DJANGO_SECRET_KEY
[2024-02-26 22:19:15 +0000] [7] [INFO] Worker exiting (pid: 7)
[2024-02-26 22:19:15 +0000] [1] [ERROR] Worker (pid:7) exited with code 3
[2024-02-26 22:19:15 +0000] [1] [ERROR] Shutting down: Master
[2024-02-26 22:19:15 +0000] [1] [ERROR] Reason: Worker failed to boot.
```

Oh, whoops; it can't find the `DJANGO_SECRET_KEY` environment variable. We need to set those environment variables on the server too.

Setting Environment Variables and Secrets

When we run our container manually locally with `docker run`, we can pass in environment variables with the `-e` flag. As we'll see, it's fairly straightforward to replicate that with Ansible, using the `env` parameter for the `docker.docker_container` module that we're already using.

But there is at least one "secret" value that we don't want to hardcode into our Ansible YAML file: the Django `SECRET_KEY` setting.

There are many different ways of dealing with secrets; different cloud providers have their own tools. There's also HashiCorp Vault—it has varying levels of complexity and security.

We don't have time to go into detail on those in this book. Instead, we'll generate a one-off secret key value from a random string, and we'll store it to a file on disk on the server. That's a reasonable amount of security for our purposes.

So, here's the plan:

1. We generate a random, one-off secret key the first time we deploy to a new server, and we store it in a file on disk.
2. We read the secret key value back from that file to put it into the container's environment variables.
3. We set the rest of the env vars we need as well.

Here's what it looks like:

infra/deploy-playbook.yaml (ch12l005)

```
- name: Import container image on server
  [...]

- name: Ensure .secret-key file exists
  # the intention is that this only happens once per server
  ansible.builtin.copy:  ❶
    dest: ~/.secret-key
    content: "{{ lookup('password', '/dev/null length=32 chars=ascii_letters') }}"  ❷
    mode: 0600
    force: false  # do not recreate file if it already exists.

- name: Read secret key back from file
  ansible.builtin.slurp:  ❸
    src: ~/.secret-key
  register: secret_key

- name: Run container
  community.docker.docker_container:
    name: superlists
    image: superlists
    state: started
    recreate: true
    env:  ❹
      DJANGO_DEBUG_FALSE: "1"
      DJANGO_SECRET_KEY: "{{ secret_key.content | b64decode }}"  ❺
      DJANGO_ALLOWED_HOST: "{{ inventory_hostname }}"  ❻
      DJANGO_DB_PATH: "/home/nonroot/db.sqlite3"
```

❶ The `builtin.copy` module can be used to copy local files up to the server, and also, as we're demonstrating here, to populate a file with an arbitrary string `content`.

❷ This `lookup('password')` thing is how we'll get a random string of characters. I copy-pasted it from Stack Overflow. Come on; there's no shame in that. The rest of the `builtin.copy` directive is designed to save the value to disk, but only if the file doesn't already exist. The `0600` permission will ensure that only the "elspeth" user can read it.

❸ The `slurp` command reads the contents of a file on the server, and we can `register` its contents into a variable. Slightly annoyingly, it uses base64 encoding (it's so you can also use it to read binary files). Anyway, the idea is, even though we don't *rewrite* the file on every deploy, we do *reread* the value on every deploy.

❹ Here's the `env` parameter for our container.

❺ Here's how we get our original value for the secret key, using the `| b64decode` to decode it back to a regular string.

❻ `inventory_hostname` represents the hostname of the current server we're deploying to, so *staging.ottg.co.uk* in our case.

Let's run this latest version of our playbook now:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -v
[...]
PLAYBOOK: deploy-playbook.yaml *************************************************
1 plays in infra/deploy-playbook.yaml

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
ok: [staging.ottg.co.uk]

TASK [Install docker] **********************************************************
ok: [staging.ottg.co.uk] => {"cache_update_time": 1709136057, "cache_updated":
false, "changed": false}

TASK [Build container image locally] *******************************************
changed: [staging.ottg.co.uk -> 127.0.0.1] => {"actions": ["Built image [...]

TASK [Export container image locally] ******************************************
changed: [staging.ottg.co.uk -> 127.0.0.1] => {"actions": ["Archived image [...]

TASK [Upload image to server] **************************************************
changed: [staging.ottg.co.uk] => {"changed": true, [...]

TASK [Import container image on server] ****************************************
changed: [staging.ottg.co.uk] => {"actions": ["Loaded image [...]

TASK [Ensure .env file exists] *************************************************
changed: [staging.ottg.co.uk] => {"changed": true, [...]

TASK [Run container] ***********************************************************
changed: [staging.ottg.co.uk] => {"changed": true, "container": [...]

PLAY RECAP *********************************************************************
staging.ottg.co.uk         : ok=8    changed=6    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

Manually Checking Environment Variables for Running Containers

We'll do one more manual check with SSH, to see if those env vars were set correctly. There's a couple of ways we can do this.

Let's start with a `docker ps` to check whether our container is running:

```
elspeth@server:$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS
PORTS     NAMES
96d867b42a31   superlists   "gunicorn --bind :88…"   6 seconds ago   Up 5
seconds             superlists
```

Looking good! The `STATUS: Up 5 Seconds` is better than the `Exited` we had before; that means the container is up and running.

Let's take a look at the `docker logs` too:

```
elspeth@server:~$ docker logs superlists
[2025-05-02 17:55:18 +0000] [1] [INFO] Starting gunicorn 23.0.0
[2025-05-02 17:55:18 +0000] [1] [INFO] Listening at: http://0.0.0.0:8888 (1)
[2025-05-02 17:55:18 +0000] [1] [INFO] Using worker: sync
[2025-05-02 17:55:18 +0000] [7] [INFO] Booting worker with pid: 7
```

Also looking good; no sign of an error. Now let's check on those environment variables. There are two ways we can do this: `docker exec env` and `docker inspect`.

docker exec env

One way is to run the standard shell `env` command, which prints out all environment variables. We run it "inside" the container with `docker exec`:

```
elspeth@server:~$ docker exec superlists env
PATH=/venv/bin:/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=96d867b42a31
DJANGO_DEBUG_FALSE=1
DJANGO_SECRET_KEY=cXACJZTvoPfWFSBSTdixJTlXCWYTnJlC
DJANGO_ALLOWED_HOST=staging.ottg.co.uk
DJANGO_DB_PATH=/home/nonroot/db.sqlite3
GPG_KEY=7169605F62C751356D054A26A821E680E5FA6305
PYTHON_VERSION=3.14.3
PYTHON_SHA256=40f868bcbdeb8149a3149580bb9bfd407b3321cd48f0be631af955ac92c0e041
HOME=/home/nonroot
```

docker inspect

Another option—useful for debugging other things too, like image IDs and mounts—is to use `docker inspect`:

```
elspeth@server:~$ docker inspect superlists
[
    {
        [...]
        "Config": {
            [...]
            "Env": [
                "DJANGO_DEBUG_FALSE=1",
                "DJANGO_SECRET_KEY=cXACJZTvoPfWFSBSTdixJTlXCWYTnJlC",
                "DJANGO_ALLOWED_HOST=staging.ottg.co.uk",
                "DJANGO_DB_PATH=/home/nonroot/db.sqlite3",
                "PATH=/venv/bin:/usr/local/bin:/usr/local/sbin:/usr/[...]
                "GPG_KEY=7169605F62C751356D054A26A821E680E5FA6305",
                "PYTHON_VERSION=3.14.3",
                "PYTHON_SHA256=40f868bcbdeb8149a3149580bb9bfd407b332[...]
            ],
            "Cmd": [
                "gunicorn",
                "--bind",
                ":8888",
                "superlists.wsgi:application"
            ],
            "Image": "superlists",
            "Volumes": null,
            "WorkingDir": "/src",
            "Entrypoint": null,
            "OnBuild": null,
            "Labels": {}
        },
        "NetworkSettings": {
          [...]
        }
    }
]
```

There's a lot of output! It's more or less everything that Docker knows about the container. But if you scroll around, you can usually get some useful info for debugging and diagnostics—like, in this case, the `Env` parameter which tells us what environment variables were set for the container.

`docker inspect` is also useful for checking exactly which image ID a container is using, and which filesystem mounts are configured.

Looking good!

Running FTs to Check on Our Deploy

Enough manual checking via SSH; let's see what our tests think. The `TEST_SERVER` adaptation we made in Chapter 9 can also be used to check against our staging server.

Let's see what they think:

```
$ TEST_SERVER=staging.ottg.co.uk python src/manage.py test functional_tests
[...]
selenium.common.exceptions.WebDriverException: Message: Reached error page:
about:neterror?e=connectionFailure&u=http%3A//staging.ottg.co.uk/[...]
[...]
Ran 3 tests in 5.014s

FAILED (errors=3)
```

None of them passed. Hmm. That `neterror` makes me think it's another networking problem.

If your domain provider puts up a temporary holding page, you may get a 404 rather than a connection error at this point, and the traceback might have "NoSuchElementException" instead.

Manual Debugging with curl Against the Staging Server

Let's try our standard debugging technique of using `curl` both locally and then from inside the container on the server. First, on our own machine:

```
$ curl -iv staging.ottg.co.uk
[...]
curl: (7) Failed to connect to staging.ottg.co.uk port 80 after 25 ms: Couldn't
connect to server
```

Similarly, depending on your domain/hosting provider, you may see "Host not found" here instead. Or, if your version of `curl` is different, you might see "Connection refused".

Now let's SSH in to our server and take a look at the Docker logs:

```
elspeth@server$ docker logs superlists
[2024-02-28 22:14:43 +0000] [7] [INFO] Starting gunicorn 21.2.0
[2024-02-28 22:14:43 +0000] [7] [INFO] Listening at: http://0.0.0.0:8888 (7)
[2024-02-28 22:14:43 +0000] [7] [INFO] Using worker: sync
[2024-02-28 22:14:43 +0000] [8] [INFO] Booting worker with pid: 8
```

No errors there. Let's try our `curl`:

```
elspeth@server$ curl -iv localhost
*   Trying 127.0.0.1:80...
* connect to 127.0.0.1 port 80 failed: Connection refused
*   Trying ::1:80...
* connect to ::1 port 80 failed: Connection refused
* Failed to connect to localhost port 80 after 0 ms: Connection refused
* Closing connection 0
curl: (7) Failed to connect to localhost port 80 after 0 ms: Connection refused
```

Hmm, `curl` fails on the server too. But all this talk of port `80`, both locally and on the server, might be giving us a clue. Let's check `docker ps`:

```
elspeth@server:$ docker ps
CONTAINER ID   IMAGE        COMMAND                  CREATED         STATUS
PORTS     NAMES
1dd87cbfa874   superlists   "/bin/sh -c 'gunicor…"   9 minutes ago   Up 9
minutes             superlists
```

This might be ringing a bell now—we forgot the ports.

We want to map port `8888` inside the container as port `80` (the default web/HTTP port) on the server:

infra/deploy-playbook.yaml (ch12l006)

```
    - name: Run container
      community.docker.docker_container:
        name: superlists
        image: superlists
        state: started
        recreate: true
        env:
          DJANGO_DEBUG_FALSE: "1"
          DJANGO_SECRET_KEY: "{{ secret_key.content | b64decode }}"
          DJANGO_ALLOWED_HOST: "{{ inventory_hostname }}"
          DJANGO_DB_PATH: "/home/nonroot/db.sqlite3"
        ports: 80:8888
```

You can map a different port on the outside to the one that's "inside" the Docker container. In this case, we can map the public-facing standard HTTP port `80` on the host to the arbitrarily chosen port `8888` on the inside.

Let's push that up with `ansible-playbook`:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, \
  infra/deploy-playbook.yaml -v
[...]
```

And now give the FTs another go:

```
$ TEST_SERVER=staging.ottg.co.uk python src/manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_list_table"]; [...]
[...]
Ran 3 tests in 21.047s

FAILED (errors=3)
```

So, 3/3 failed again, but the FTs *did* get a little further along. If you saw what was happening, or if you go and visit the site manually in your browser, you'll see that the home page loads fine, but as soon as we try and create a new list item, it crashes with a 500 error.

Mounting the Database on the Server and Running Migrations

Let's do another bit of manual debugging, and take a look at the logs from our container with `docker logs`. You'll see an `OperationalError`:

```
$ ssh elspeth@server docker logs superlists
[...]
django.db.utils.OperationalError: no such table: lists_list
```

It looks like our database isn't initialised. Aha! Another of those deployment "danger areas".

Just like we did on our own machine, we need to mount the `db.sqlite3` file from the filesystem outside the container. We'll also want to run migrations to create the database and, in fact, each time we deploy, so that any updates to the database schema get applied to the database on the server.

Here's the plan:

1. On the host machine, we'll store the database in elspeth's home folder; it's as good a place as any.
2. We'll set its UID to `1234`, just like we did in Chapter 10, to match the UID of the `nonroot` user inside the container.
3. Inside the container, we'll use the path `/home/nonroot/db.sqlite3`—again, just like in the last chapter.
4. We'll run the migrations with a `docker exec`, or the Ansible equivalent thereof.

Here's what that looks like:

infra/deploy-playbook.yaml (ch12l007)

```
- name: Ensure db.sqlite3 file exists outside container
  ansible.builtin.file:
    path: "{{ ansible_env.HOME }}/db.sqlite3"  ❶
    state: touch  ❷
    owner: 1234  # so nonroot user can access it in container
  become: true  # needed for ownership change

- name: Run container
  community.docker.docker_container:
    name: superlists
    image: superlists
    state: started
    recreate: true
    env:
      DJANGO_DEBUG_FALSE: "1"
      DJANGO_SECRET_KEY: "{{ secret_key.content | b64decode }}"
      DJANGO_ALLOWED_HOST: "{{ inventory_hostname }}"
      DJANGO_DB_PATH: "/home/nonroot/db.sqlite3"
    mounts:  ❸
      - type: bind
        source: "{{ ansible_env.HOME }}/db.sqlite3"  ❶
        target: /home/nonroot/db.sqlite3
    ports: 80:8888

- name: Run migration inside container
  community.docker.docker_container_exec:  ❹
    container: superlists
    command: ./manage.py migrate
```

❶ `ansible_env` gives us access to the environment variables on the server, including `HOME`, which is the path to the home folder (*/home/elspeth/* in my case).

❷ We use `file` with `state=touch` to make sure a placeholder file exists before we try and mount it in.

❸ Here is the `mounts` config, which works a lot like the `--mount` flag to `docker run`.

❹ And we use the `docker.container_exec` module to give us the functionality of `docker exec`, to run the migration command inside the container.

Let's give that playbook a run and…

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -v
[...]
TASK [Run migration inside container] ******************************************
changed: [staging.ottg.co.uk] => {"changed": true, "rc": 0, "stderr": "",
"stderr_lines": [], "stdout": "Operations to perform:\n  Apply all migrations:
auth, contenttypes, lists, sessions\nRunning migrations:\n  Applying
contenttypes.0001_initial... OK\n  Applying
contenttypes.0002_remove_content_type_name... OK\n  Applying
auth.0001_initial... OK\n  Applying
auth.0002_alter_permission_name_max_length... OK\n  Applying
[...]
PLAY RECAP *********************************************************************
staging.ottg.co.uk         : ok=9    changed=2    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

It Workssss

Try the tests…

```
$ TEST_SERVER=staging.ottg.co.uk python src/manage.py test functional_tests
Found 3 test(s).
[...]

...
 ---------------------------------------------------------------------
Ran 3 tests in 13.537s
OK
```

Hooray!

All the tests pass! That gives us confidence that our automated deploy script can reproduce a fully working app, on a server, hosted on the public internet.

That's worthy of a commit:

```
$ git diff
# should show our changes in deploy-playbook yaml
$ git commit -am"Save secret key, set env vars, mount db, run migrations. It works :)"
```

Deploying to Prod

Now that we are confident in our deploy script, let's try using it for our live site!

The main change is to the `-i` flag, where we pass in the production domain name, instead of the staging one:

```
$ ansible-playbook --user=elspeth -i www.ottg.co.uk, infra/deploy-playbook.yaml -vv
[...]

Done.
Disconnecting from elspeth@www.ottg.co.uk... done.
```

Brrp brrp brpp. Looking good? Go take a click around your live site!

git tag the Release

One final bit of admin. To preserve a historical marker, we'll use Git tags to mark the state of the codebase that reflects what's currently live on the server:

```
$ git tag LIVE
$ export TAG=$(date +DEPLOYED-%F/%H%M)  # this generates a timestamp
$ echo $TAG # should show "DEPLOYED-" and then the timestamp
$ git tag $TAG
$ git push origin LIVE $TAG # pushes the tags up to GitHub
```

Now it's easy, at any time, to check what the difference is between our current codebase and what's live on the servers. This will come in handy in a few chapters, when we look at database migrations. Have a look at the tag in the history:

```
$ git log --graph --oneline --decorate
* 1d4d814 (HEAD -> main) Save secret key, set env vars, mount db, run
migrations. It works :)
* 95e0fe0 Build our image, use export/import to get it on the server, try and
run it
* 5a36957 Made a start on an ansible playbook for deployment
[...]
```

Once again, this use of Git tags isn't meant to be the *one true way*. We just need some sort of way to keep track of what was deployed when.

Tell Everyone!

You now have a live website! Tell all your friends! Tell your mum, if no one else is interested! Or, tell me at *obeythetestinggoat@gmail.com*! I'm always delighted to see a new reader's site!

Congratulations again for getting through this block of deployment chapters; I know they can be challenging. I hope you got something out of them—seeing a practical example of how to take these kinds of complex changes and break them down into small, incremental steps, getting frequent feedback from our tests and manual investigations along the way.

Our next deploy won't be until Chapter 18, so you can switch off your servers until then if you want to. If you're using a platform where you only get one month of free hosting, it might run out by then. You might have to shell out a few bucks, or see if there's some way of getting another free month.

In the next chapter, it's back to coding again.

Further Reading

There's no such thing as the *one true way* in deployment; I've tried to set you off on a reasonably sane path, but there are plenty of things you could do differently—and lots, *lots* more to learn besides. Here are some resources I used for inspiration, (including a couple I've already mentioned):

- The original Twelve-Factor App manifesto from the Heroku team
- The official Django docs' Deployment Checklist
- "How to Write Deployment-friendly Applications" by Hynek Schlawack
- The deployment chapter of *Two Scoops of Django* by Daniel and Audrey Roy Greenfield
- The PythonSpeed "Docker packaging for Python developers" guide

Automated Deployment and IaC Recap

Here's a brief recap of what we've been through, which are a fairly typical set of steps for deployment in general:

Provisioning a server
: This tends to be vendor-specific, so we didn't automate it, but you absolutely can!

Installing system dependencies
: In our case, it was mainly Docker. But inside the Docker image, we also had some system dependencies too, like Python itself. The installation of both types of dependencies is now automated, and now defined "in code", whether it's the Dockerfile or the Ansible YAML.

Getting our application code (or "artifacts") onto the server
: In our case, because we're using Docker, the thing we needed to transfer was a Docker image. Typically, you would do this by pushing and pulling from an image repository—although in our automation, we used a more direct process, purely to avoid endorsing any particular vendor.

Setting environment variables and secrets
: Depending on how you need to vary them, you can set environment variables on your local PC, in a Dockerfile, in your Ansible scripts, or on the server itself. Figuring out which to use in which case is a big part of deployment.

Attaching to the database
: In our case, we mount a file from the local filesystem. More typically, you'd be supplying some environment variables and secrets to define a host, port, username, and password to use for accessing a database server.

Configuring networking and port mapping
: This includes DNS config, as well as Docker configuration. Web apps need to be able to talk to the outside world!

Running database migrations
: We'll revisit this later in the book, but migrations are one of the most risky parts of a deployment, and automating them is a key part of reducing that risk.

Going live with the new version of our application
: In our case, we stop the old container and start a new one. In more advanced setups, you might be trying to achieve zero downtime deploys, and looking into techniques like blue/green deployments, but those are topics for different books.

Every single aspect of deployment can and probably should be automated. Here are a couple of general principles to think about when implementing IaC:

Idempotence
: If your deployment script is deploying to existing servers, you need to design them so that they work against a fresh installation *and* against a server that's already configured.

Declarative
: As much as possible, we want to try and specify *what* we want the state to be on the server, rather than *how* we should get there. This goes hand in hand with the idea of idempotence.

PART III
Forms and Validation

Now that we've got things into production, we'll spend a bit of time on validation, a core topic in web development.

There's quite a lot of Django-specific content in this part, so if you weren't familiar with Django before starting on the book, you may find that taking a little time to run through the official Django tutorial will complement the next few chapters nicely.

With that said, there are lots of good lessons about test-driven development (TDD) in general in here too! So, alternatively, if you're not that interested in Django itself, don't worry too much about the details; instead, look out for the more general principles of testing.

Here's a little preview of what we'll cover:

- Splitting tests out across multiple files
- Using a decorator for Selenium waits/polling
- Database-layer validation and constraints
- HTML5 form validation in the frontend
- The Django forms framework
- The trade-offs of frameworks in general, and when to stop using them
- How far to go when testing for possible coding errors
- An overview of all the typical tests for Django views

CHAPTER 13

Splitting Our Tests into Multiple Files, and a Generic Wait Helper

Back to local development! The next feature we might like to implement is a little input validation. But as we start writing new tests, we'll notice that it's getting hard to find our way around a single *functional_tests.py*, and *tests.py*, so we'll reorganise them into multiple files—a little refactor of our tests, if you will.

We'll also build a generic explicit wait helper.

Start on a Validation FT: Preventing Blank Items

As our first few users start using the site, we've noticed they sometimes make mistakes that mess up their lists, like accidentally submitting blank list items, or inputting two identical items to a list. Computers are meant to help stop us from making silly mistakes, so let's see if we can get our site to help.

Here's the outline of the new FT method, which we will add to `NewVisitorTestCase`:

src/functional_tests/tests.py (ch13l001)

```
def test_cannot_add_empty_list_items(self):
    # Edith goes to the home page and accidentally tries to submit
    # an empty list item. She hits Enter on the empty input box

    # The home page refreshes, and there is an error message saying
    # that list items cannot be blank

    # She tries again with some text for the item, which now works

    # Perversely, she now decides to submit a second blank list item

    # She receives a similar warning on the list page

    # And she can correct it by filling some text in
    self.fail("write me!")
```

That's all very well, but before we go any further—our functional tests (FTs) file is beginning to get a little crowded. Let's split it out into several files, in which each has a single test method.

Remember that FTs are closely linked to "user stories" and features. One way of organising your FTs might be to have one per high-level feature.

We'll also have one base test class, which they can all inherit from. Here's how to get there step by step.

Skipping a Test

We're back to local development now. Make sure that the `TEST_SERVER` environment variable is unset by executing the command `unset TEST_SERVER` from the terminal.

It's always nice, when refactoring, to have a fully passing test suite. We've just written a test with a deliberate failure. Let's temporarily switch it off, using a decorator called "skip" from `unittest`:

src/functional_tests/tests.py (ch13l001-1)

```
from unittest import skip
[...]

    @skip
    def test_cannot_add_empty_list_items(self):
```

This tells the test runner to ignore this test. You can see it works—if we rerun the tests, you'll see it's a pass, but it explicitly mentions the skipped test:

```
$ python src/manage.py test functional_tests
[...]
Ran 4 tests in 11.577s
OK (skipped=1)
```

Skips are dangerous—you need to remember to remove them before you commit your changes back to the repo. This is why line-by-line reviews of each of your diffs are a good idea!

Don't Forget the "Refactor" in "Red/Green/Refactor"

A criticism that's sometimes levelled at TDD is that it leads to badly architected code, as the developer just focuses on getting tests to pass rather than stopping to think about how the whole system should be designed. I think it's slightly unfair.

TDD is no silver bullet. You still have to spend time thinking about good design. But what often happens is that people forget the "refactor" in "red/green/refactor". The methodology allows you to throw together any old code to get your tests to pass, but it *also* asks you to then spend some time refactoring it to improve its design. Otherwise, it's too easy to allow "technical debt" to build up.

Often, however, the best ideas for how to refactor code don't occur to you straight away. They may occur to you days, weeks, even months after you wrote a piece of code, when you're working on something totally unrelated and you happen to see some old code again with fresh eyes. But if you're halfway through something else, should you stop to refactor the old code?

The answer is that it depends. In the case at the beginning of the chapter, we haven't even started writing our new code. We know we are in a working state, so we can justify putting a skip on our new FT (to get back to fully passing tests) and do a bit of refactoring straight away.

Later in the chapter, we'll spot other bits of code we want to alter. In those cases, rather than taking the risk of refactoring an application that's not in a working state, we'll make a note of the thing we want to change on our scratchpad and wait until we're back to a fully passing test suite before refactoring.

Kent Beck has a book-length exploration of the trade-offs of refactor-now versus refactor-later, called *Tidy First?*.

Splitting Functional Tests Out into Many Files

We start putting each test into its own class, still in the same file:

src/functional_tests/tests.py (ch13l002)

```
class FunctionalTest(StaticLiveServerTestCase):
    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def wait_for_row_in_list_table(self, row_text):
        [...]


class NewVisitorTest(FunctionalTest):
    def test_can_start_a_todo_list(self):
        [...]
    def test_multiple_users_can_start_lists_at_different_urls(self):
        [...]


class LayoutAndStylingTest(FunctionalTest):
    def test_layout_and_styling(self):
        [...]


class ItemValidationTest(FunctionalTest):
    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

At this point, we can rerun the FTs and see they all still work:

```
Ran 4 tests in 11.577s

OK (skipped=1)
```

That's labouring it a little bit, and we could probably get away with doing this stuff in fewer steps, but—as I keep saying—practising the step-by-step method on the easy cases makes it that much easier when we have a complex case.

Now we switch from a single tests file to using one for each class, and one "base" file to contain the base class that all the tests will inherit from. We'll make four copies of *tests.py*, naming them appropriately, and then delete the parts we don't need from each:

```
$ git mv src/functional_tests/tests.py src/functional_tests/base.py
$ cp src/functional_tests/base.py src/functional_tests/test_simple_list_creation.py
$ cp src/functional_tests/base.py src/functional_tests/test_layout_and_styling.py
$ cp src/functional_tests/base.py src/functional_tests/test_list_item_validation.py
```

base.py can be cut down to just the `FunctionalTest` class. We leave the helper method on the base class, because we suspect we're about to reuse it in our new FT:

src/functional_tests/base.py (ch13l003)

```
import os
import time

from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium import webdriver
from selenium.common.exceptions import WebDriverException
from selenium.webdriver.common.by import By

MAX_WAIT = 5


class FunctionalTest(StaticLiveServerTestCase):
    def setUp(self):
        [...]
    def tearDown(self):
        [...]
    def wait_for_row_in_list_table(self, row_text):
        [...]
```

Keeping helper methods in a base `FunctionalTest` class is one useful way of preventing duplication in FTs. Later in the book (in Chapter 26), we'll use the "page pattern", which is related, but prefers composition over inheritance—always a good thing.

Our first FT is now in its own file, and should be just one class and one test method:

src/functional_tests/test_simple_list_creation.py (ch13l004)

```
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

from .base import FunctionalTest


class NewVisitorTest(FunctionalTest):
    def test_can_start_a_todo_list(self):
        [...]
    def test_multiple_users_can_start_lists_at_different_urls(self):
        [...]
```

I used a relative import (`from .base`). Some people like to use them a lot in Django code (e.g., your views might import models using `from .models import List`, instead of `from list.models`). Ultimately, this is a matter of personal preference. I prefer to use relative imports only when I'm super, super confident that the relative

position of the thing I'm importing won't change. That applies in this case because I know for sure that all the tests will sit next to *base.py*, which they inherit from.

The layout and styling FT should now be one file and one class:

src/functional_tests/test_layout_and_styling.py (ch13l005)

```
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

from .base import FunctionalTest


class LayoutAndStylingTest(FunctionalTest):
        [...]
```

Lastly, our new validation test is in a file of its own too:

src/functional_tests/test_list_item_validation.py (ch13l006)

```
from unittest import skip

from selenium.webdriver.common.by import By  ❶
from selenium.webdriver.common.keys import Keys  ❶

from .base import FunctionalTest


class ItemValidationTest(FunctionalTest):
    @skip
    def test_cannot_add_empty_list_items(self):
        [...]
```

❶ These two will be marked as "unused imports" for now but that's OK; we'll use them shortly.

And we can test that everything worked by rerunning `manage.py test functional_tests`, and checking once again that all four tests are run:

```
Ran 4 tests in 11.577s

OK (skipped=1)
```

Now we can remove our skip:

src/functional_tests/test_list_item_validation.py (ch13l007)

```
class ItemValidationTest(FunctionalTest):
    def test_cannot_add_empty_list_items(self):
        [...]
```

Running a Single Test File

As a side bonus, we're now able to run an individual test file, like this:

```
$ python src/manage.py test functional_tests.test_list_item_validation
[...]
AssertionError: write me!
```

Brilliant—no need to sit around waiting for all the FTs when we're only interested in a single one. Although, we need to remember to run all of them now and again to check for regressions. Later in the book, we'll set up a continuous integration (CI) server to run all the tests automatically—for example, every time we push to the main branch. For now, a good prompt for running all the tests is "just before you do a commit", so let's get into that habit now:

```
$ git status
$ git add src/functional_tests
$ git commit -m "Moved FTs into their own individual files"
```

Great. We've split our FTs nicely out into different files. Next, we'll start writing our FT. But before long, as you may be guessing, we'll do something similar to our unit test files.

A New FT Tool: A Generic Explicit Wait Helper

First, let's start implementing the test—or at least the beginning of it:

src/functional_tests/test_list_item_validation.py (ch13l008)

```
def test_cannot_add_empty_list_items(self):
    # Edith goes to the home page and accidentally tries to submit
    # an empty list item. She hits Enter on the empty input box
    self.browser.get(self.live_server_url)
    self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)

    # The home page refreshes, and there is an error message saying
    # that list items cannot be blank
    self.assertEqual(
        self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback").text,  ❶
        "You can't have an empty list item",  ❷
    )

    # She tries again with some text for the item, which now works
    self.fail("finish this test!")
    [...]
```

This is how we might write the test naively:

❶ We specify we're going to use a CSS class called `.invalid-feedback` to mark our error text. We'll see that Bootstrap has some useful styling for those.

❷ And we can check that our error displays the message we want.

But can you guess what the potential problem is with the test as it's written now?

OK, I gave it away in the section header, but whenever we do something that causes a page refresh, we need an explicit wait; otherwise, Selenium might go looking for the `.invalid-feedback` element before the page has had a chance to load.

Whenever you submit a form with `Keys.ENTER` or click something that is going to cause a page to load, you probably want an explicit wait for your next assertion.

Our first explicit wait was built into a helper method. For this one, we might decide that building a specific helper method is overkill at this stage, but it might be nice to have some generic way of saying in our tests, "wait until this assertion passes". Something like this:

src/functional_tests/test_list_item_validation.py (ch13l009)

```
[...]
    # The home page refreshes, and there is an error message saying
    # that list items cannot be blank
    self.wait_for(
        lambda: self.assertEqual(  ❶
            self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback").text,
            "You can't have an empty list item",
        )
    )
```

❶ Rather than calling the assertion directly, we wrap it in a `lambda` function, and we pass it to a new helper method we imagine called `wait_for`.

If you've never seen `lambda` functions in Python before, see "Lambda Functions" on page 294.

So, how would this magical `wait_for` method work? Let's head over to *base.py*, make a copy of our existing `wait_for_row_in_list_table` method, and we'll adapt it slightly:

src/functional_tests/base.py (ch13l010)

```
def wait_for(self, fn):  ❶
    start_time = time.time()
    while True:
        try:
            table = self.browser.find_element(By.ID, "id_list_table")  ❷
            rows = table.find_element(By.TAG_NAME, "tr")
            self.assertIn(row_text, [row.text for row in rows])
            return
        except (AssertionError, WebDriverException):
            if time.time() - start_time > MAX_WAIT:
                raise
            time.sleep(0.5)
```

❶ We make a copy of the method, but we name it `wait_for`, and we change its argument. It is expecting to be passed a function.

❷ For now, we've still got the old code that's checking table rows. Now, how do we transform this into something that works for any generic `fn` that's been passed in?

Like this:

src/functional_tests/base.py (ch13l011)

```
def wait_for(self, fn):
    start_time = time.time()
    while True:
        try:
            return fn()  ❶
        except (AssertionError, WebDriverException):
            if time.time() - start_time > MAX_WAIT:
                raise
            time.sleep(0.5)
```

❶ The body of our `try/except`, instead of being the specific code for examining table rows, just becomes a call to the function we passed in. We also `return` its result, to be able to exit the loop immediately if no exception is raised.

Lambda Functions

`lambda` in Python is the syntax for making a one-line, throwaway function. It saves you from having to use `def...():` and an indented block:

```
>>> myfn = lambda x: x+1
>>> myfn(2)
3
>>> myfn(5)
6
>>> adder = lambda x, y: x + y
>>> adder(3, 2)
5
```

In our case, we're using it to transform a bit of code, that would otherwise be executed immediately, into a function that we can pass as an argument, and that can be executed later, and multiple times:

```
>>> def addthree(x):
...     return x + 3
...
>>> addthree(2)
5
>>> myfn = lambda: addthree(2)  # note addthree isn't called immediately here
>>> myfn
<function <lambda> at 0x7f3b140339d8>
>>> myfn()
5
>>> myfn()
5
```

Let's see our funky `wait_for` helper in action:

```
$ python src/manage.py test functional_tests.test_list_item_validation
[...]

======================================================================
ERROR: test_cannot_add_empty_list_items (functional_tests.test_list_item_valida
tion.ItemValidationTest.test_cannot_add_empty_list_items)
 ----------------------------------------------------------------------
[...]
Traceback (most recent call last):
  File "...goat-book/src/functional_tests/test_list_item_validation.py", line
16, in test_cannot_add_empty_list_items
    self.wait_for(❶
  File "...goat-book/src/functional_tests/base.py", line 25, in wait_for
    return fn()❷
           ^^^^
  File "...goat-book/src/functional_tests/test_list_item_validation.py", line
18, in <lambda>❸
    self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback").text,❸
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; [...]

 ----------------------------------------------------------------------
Ran 1 test in 10.575s

FAILED (errors=1)
```

The order of the traceback is a little confusing, but we can more or less follow through what happened:

❶ In our FT, we call our `self.wait_for` helper, where we pass the `lambda`-ified version of `assertEqual`.

❷ We go into `self.wait_for` in *base.py*, where we're calling (and returning) `fn()`, which refers to the passed `lambda` function encapsulating our test assertion.

❸ To explain where the exception has actually come from, the traceback takes us back into *test_list_item_validation.py* and inside the body of the `lambda` function, and tells us that it was attempting to find the `.invalid-feedback` element that failed.

We're into the realm of functional programming now, passing functions as arguments to other functions, and it can be a little mind-bending. I know it took me a little while to get used to! Have a couple of read-throughs of this code, and the code back in the FT, to let it sink in; and if you're still confused, don't worry about it too much, and let your confidence grow from working with it. We'll use it a few more times in this book and make it even more functionally fun; you'll see.

Finishing Off the FT

We'll finish off the FT like this:

src/functional_tests/test_list_item_validation.py (ch13l012)

```
        # The home page refreshes, and there is an error message saying
        # that list items cannot be blank
        self.wait_for(
            lambda: self.assertEqual(
                self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback").text,
                "You can't have an empty list item",
            )
        )

        # She tries again with some text for the item, which now works
        self.browser.find_element(By.ID, "id_new_item").send_keys("Purchase milk")
        self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table("1: Purchase milk")

        # Perversely, she now decides to submit a second blank list item
        self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)

        # She receives a similar warning on the list page
        self.wait_for(
            lambda: self.assertEqual(
                self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback").text,
                "You can't have an empty list item",
            )
        )

        # And she can correct it by filling some text in
        self.browser.find_element(By.ID, "id_new_item").send_keys("Make tea")
        self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table("2: Make tea")
```

Helper Methods in FTs

We've got two helper methods now: our generic `self.wait_for` helper, and `wait_for_row_in_list_table`. The former is a general utility—any of our FTs might need to do a wait.

The latter also helps prevent duplication across your FT code. The day we decide to change the implementation of how our list table works, we want to make sure we only have to change our FT code in one place, not in dozens of places across loads of FTs…

See also Chapter 26 and Online Appendix: BDD for more on structuring your FT code.

I'll let you do your own "first-cut FT" commit.

Refactoring Unit Tests into Several Files

When we (finally!) start coding our solution, we're going to want to add another test for our *models.py*. Before we do so, it's time to tidy up our unit tests in a similar way to the functional tests.

A difference will be that, because the `lists` app contains real application code as well as tests, we'll separate out the tests into their own folder:

```
$ mkdir src/lists/tests
$ touch src/lists/tests/__init__.py
$ git mv src/lists/tests.py src/lists/tests/test_all.py
$ git status
$ git add src/lists/tests
$ python src/manage.py test lists
[...]
Ran 10 tests in 0.034s

OK
$ git commit -m "Move unit tests into a folder with single file"
```

If you get a message saying "Ran 0 tests", you probably forgot to add the dunderinit.[1] It needs to be there for the tests folder to be recognised as a regular Python package,[2] and thus discovered by the test runner.

Now we turn *test_all.py* into two files—one called *test_views.py*, which will only contain view tests, and one called *test_models.py*. I'll start by making two copies:

```
$ git mv src/lists/tests/test_all.py src/lists/tests/test_views.py
$ cp src/lists/tests/test_views.py src/lists/tests/test_models.py
```

1 "Dunder" is shorthand for double-underscore, so "dunderinit" means *__init__.py*.

2 Without the dunderinit, a folder with Python files in it is called a namespace package. Usually, they are exactly the same as regular packages (which *do* have a *__init__.py*), but the Django test runner does not recognise them.

And strip *test_models.py* down to being just the one test:

src/lists/tests/test_models.py (ch13l016)

```
from django.test import TestCase

from lists.models import Item, List


class ListAndItemModelsTest(TestCase):
        [...]
```

Whereas *test_views.py* just loses one class:

src/lists/tests/test_views.py (ch13l017)

```
--- a/src/lists/tests/test_views.py
+++ b/src/lists/tests/test_views.py
33 +74,3 @@ class NewItemTest(TestCase):
         )

         self.assertRedirects(response, f"/lists/{correct_list.id}/")
-
-
-class ListAndItemModelsTest(TestCase):
-    def test_saving_and_retrieving_items(self):
[...]
```

We rerun the tests to check that everything is still there:

```
$ python src/manage.py test lists
[...]
Ran 10 tests in 0.040s

OK
```

Great! That's another small, working step:

```
$ git add src/lists/tests
$ git commit -m "Split out unit tests into two files"
```

Some people like to make their unit tests into a tests folder straight away, as soon as they start a project. That's a perfectly good idea; I just thought I'd wait until it became necessary, to avoid doing too much housekeeping all in the first chapter!

Well, that's our FTs and unit tests nicely reorganised. In the next chapter, we'll get down to some validation proper.

Tips on Organising Tests and Refactoring

Use a tests folder
: Just as you use multiple files to hold your application code, you should split your tests out into multiple files:

  - For functional tests, group them into tests for a particular feature or user story.
  - For unit tests, use a folder called *tests*, with a *__init__.py*.
  - You probably want a separate test file for each tested source code file. For Django, that's typically *test_models.py*, *test_views.py*, and *test_forms.py*.
  - Have at least a placeholder test for *every* function and class.

Don't forget the "refactor" in "red/green/refactor"
: The whole point of having tests is to allow you to refactor your code! Use them, and make your code (including your tests) as clean as you can.

Don't refactor against failing tests

- The general rule is that you shouldn't mix refactoring and behaviour change. Having green tests is our best guarantee that we aren't changing behaviour. If you start refactoring against failing tests, it becomes much harder to spot when you're accidentally introducing a regression.
- This applies strongly to unit tests. With FTs, because we often develop against red FTs anyway, it's sometimes more tempting to refactor against failing tests. My suggestion is to avoid that temptation and use an early return, so that it's 100% clear if, during a refactor, you accidentally introduce a regression that's picked up in your FTs.
- You can occasionally put a skip on a test that is testing something you haven't written yet.
- More commonly, make a note of the refactor you want to do, finish what you're working on, and do the refactor a little later when you're back to a working state.
- Don't forget to remove any skips before you commit your code! You should always review your diffs line by line to catch things like this.

Try a generic wait_for helper
: Having specific helper methods that do explicit waits is great, and it helps to make your tests readable. But you'll also often need an ad-hoc, one-line assertion or Selenium interaction that you'll want to add a wait to. `self.wait_for` does the job well for me, but you might find a slightly different pattern works for you.

CHAPTER 14

Validation at the Database Layer

Over the next few chapters, we'll talk about testing and implementing validation of user inputs.

In terms of content, there's going to be quite a lot of material here that's more about the specifics of Django, and less discussion of TDD philosophy. That doesn't mean you won't be learning anything about testing—there are plenty of little testing tidbits in here, but perhaps it's more about really getting into the swing of things, the rhythm of TDD, and how we get work done.

Once we get through these three short chapters, I've saved a bit of fun with JavaScript (!) for the end of Part II. Then it's on to Part III, where I promise we'll get right back into some of the real nitty-gritty discussions on TDD methodology—unit tests versus integration tests, mocking, and more. Stay tuned!

But for now, a little validation. Let's just remind ourselves where our FT is pointing us:

```
$ python3 src/manage.py test functional_tests.test_list_item_validation
[...]

======================================================================
ERROR: test_cannot_add_empty_list_items (functional_tests.test_list_item_valida
tion.ItemValidationTest.test_cannot_add_empty_list_items)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/functional_tests/test_list_item_validation.py", line
16, in test_cannot_add_empty_list_items
    self.wait_for(
    ~~~~~~~~~~~~~^
        lambda: self.assertEqual(
        ^^^^^^^^^^^^^^^^^^^^^^^^^
    ...<2 lines>...
        )
        ^
    )
    ^
[...]
  File "...goat-book/src/functional_tests/test_list_item_validation.py", line
18, in <lambda>
    self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback").text,
    ~~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; For documentation [...]
```

It's expecting to see an error message if the user tries to input an empty item.

Model-Layer Validation

In a web app, there are two places you can do validation: on the client side (using JavaScript or HTML5 properties, as we'll see later), and on the server side. The server side is "safer" because someone can always bypass the client side, whether it's maliciously or due to some bug.

Similarly on the server side, in Django, there are two levels at which you can do validation. One is at the model level, and the other is higher up at the forms level. I like to use the lower level whenever possible, partially because I'm a bit too fond of databases and database integrity rules, and partially because, again, it's safer—you can sometimes forget which form you use to validate input, but you're always going to use the same database.

The self.assertRaises Context Manager

Let's go down and write a unit test at the models layer. Add a new test method to `ListAndItemModelsTest`, which tries to create a blank list item. This test is interesting because it's testing that the code under test should raise an exception:

src/lists/tests/test_models.py (ch14l001)

```
from django.db.utils import IntegrityError
[...]

class ListAndItemModelsTest(TestCase):
    def test_saving_and_retrieving_items(self):
        [...]

    def test_cannot_save_empty_list_items(self):
        mylist = List.objects.create()
        item = Item(list=mylist, text="")
        with self.assertRaises(IntegrityError):
            item.save()
```

This is a new unit testing technique: when we want to check that doing something will raise an error, we can use the `self.assertRaises` context manager.

We could have used something like this instead:

```
try:
    item.save()
    self.fail('The save should have raised an exception')
except IntegrityError:
    pass
```

But the `with` formulation is neater.

If you're new to Python, you may never have seen the `with` statement. It's the special keyword to use with what are called "context managers". Together, they wrap a block of code, usually with some kind of setup, cleanup, or error-handling code. There's a good write-up on Python Morsels.

Django Model Constraints and Their Interaction with Databases

When we run this new unit test, we see the failure we expected:

```
    with self.assertRaises(IntegrityError):
AssertionError: IntegrityError not raised
```

But all is not quite as it seems, because *this test should already pass*.

If you take a look at the docs for the Django model fields, you'll see under "Field choices" that the default setting for *all* fields is `blank=False`. Because "text field" is a type of field, it *should* already disallow empty values.

So, why is the test still failing? Why is our database not raising an `IntegrityError` when we try to save an empty string into the `text` column?

The answer is a combination of Django's design and the database we're using.

Inspecting Our Constraints at the Database Level

Let's have a look directly at the database using the `dbshell` command:

```
$ ./src/manage.py dbshell  # (this is equivalent to running sqlite3 src/db.sqlite3)
SQLite version 3.[...]
Enter ".help" for usage hints.
sqlite> .schema lists_item
CREATE TABLE IF NOT EXISTS "lists_item" ("id" integer NOT NULL PRIMARY KEY
AUTOINCREMENT, "text" text NOT NULL, "list_id" bigint NOT NULL REFERENCES
"lists_list" ("id") DEFERRABLE INITIALLY DEFERRED);
```

The `text` column only has the `NOT NULL` constraint. This means that the database would not allow `None` as a value, but it will actually allow the empty string.

Whilst it is technically possible to implement a "not empty string" constraint on a text column in SQLite, the Django developers have chosen not to do this. This is because Django distinguishes between what they call "database-related" and "validation-related" constraints. As well as `empty=False`, all fields get a `null=False` setting, which translates into the database-level `NOT NULL` constraint we saw earlier.

Let's see if we can verify that using our test, instead. We'll pass in `text=None` instead of `text=""` (and change the test name):

src/lists/tests/test_models.py (ch14l002)

```
    def test_cannot_save_null_list_items(self):
        mylist = List.objects.create()
        item = Item(list=mylist, text=None)
        with self.assertRaises(IntegrityError):
            item.save()
```

You'll see that *this* test now passes:

```
Ran 11 tests in 0.030s

OK
```

Testing Django Model Validation

That's all vaguely interesting, but it's not actually what we set out to do. How do we make sure that the "validation-related" constraint is being enforced? The answer is that, while `IntegrityError` comes from the database, Django uses `ValidationError` to signal errors that come from its own validation.

Let's write a second test that checks on that:

src/lists/tests/test_models.py (ch14l003)

```
from django.core.exceptions import ValidationError
from django.db.utils import IntegrityError
[...]

class ListAndItemModelsTest(TestCase):
    def test_saving_and_retrieving_items(self):
        [...]

    def test_cannot_save_null_list_items(self):
        mylist = List.objects.create()
        item = Item(list=mylist, text=None)
        with self.assertRaises(IntegrityError):
            item.save()

    def test_cannot_save_empty_list_items(self):
        mylist = List.objects.create()
        item = Item(list=mylist, text="")  ❶
        with self.assertRaises(ValidationError):  ❷
            item.save()
```

❶ This time we pass `text=""`.

❷ And we're expecting a `ValidationError` instead of an `IntegrityError`.

A Django Quirk: Model Save Doesn't Run Validation

We can try running this new unit test, and we'll see its expected failure…

```
        with self.assertRaises(ValidationError):
AssertionError: ValidationError not raised
```

Wait a minute! We expected this to *pass* actually! We just got through learning that Django should be enforcing the `blank=False` constraint by default. Why doesn't this work?

We've discovered one of Django's little quirks. For slightly counterintuitive historical reasons, Django models don't run full validation on save.

Django does have a method to manually run full validation, however, called `full_clean` (more info in the docs). Let's swap that for the `.save()` and see if it works:

src/lists/tests/test_models.py (ch14l004)

```
        with self.assertRaises(ValidationError):
            item.full_clean()
```

That gets the unit test to pass:

```
Ran 12 tests in 0.030s

OK
```

Good. That taught us a little about Django validation, and the test is there to warn us if we ever forget our requirement and set `blank=True` on the `text` field (try it!).

Recap: Database-level and Model-level Validation in Django

Django distinguishes two types of validation for models:

1. Database-level constraints like `null=False` or `unique=True` (as we'll see an example of in Chapter 16), which are enforced by the database itself, using things like `NOT NULL` or `UNIQUE` constraints and bubble up as `IntegrityErrors` if you try to save an invalid object
2. Model-level validations like `blank=False`, which are only enforced by Django, when you call `full_clean()`, and they raise a `ValidationError`

The subtlety is that Django also enforces database-level constraints when you call `full_clean()`. So, you'll only see `IntegrityError` if you forget to call `full_clean()` before doing a `.save()`.

The FTs are still failing, because we're not actually forcing these errors to appear in our actual app, outside of this one unit test.

Surfacing Model Validation Errors in the View

Let's try to enforce our model validation in the views layer and bring it up into our templates so the user can see them. To optionally display an error in our HTML, we check whether the template has been passed an error variable and, if so, we do this:

src/lists/templates/base.html (ch14l005)

```
<form method="POST" action="{% block form_action %}{% endblock %}">
  <input
    class="form-control form-control-lg {% if error %}is-invalid{% endif %}" ❶
    name="item_text"
    id="id_new_item"
    placeholder="Enter a to-do item"
  />
  {% csrf_token %}
  {% if error %}
    <div class="invalid-feedback">{{ error }}</div> ❷
  {% endif %}
</form>
```

❶ We add the `.is-invalid` class to any form inputs that have validation errors.

❷ We use a `div.invalid-feedback` to display any error messages from the server.

Take a look at the Bootstrap docs for more info on form controls.

However, ignore the Bootstrap docs' advice to prefer client-side validation. Ideally, having both server- and client-side validation is the best. If you can't do both, then server-side validation is the one you really can't do without. Check the OWASP checklist, if you are not convinced yet. Client-side validation will provide faster feedback on the UI, but it is not a security measure. Server-side validation is indispensable for handling any input that gets processed by the server—and it will also provide (albeit slower) feedback for the client side.

Passing this error to the template is the view function's job. Let's take a look at the unit tests in the `NewListTest` class. I'm going to use two slightly different error-handling patterns here.

In the first case, our URL and view for new lists will optionally render the same template as the home page, but with the addition of an error message. Here's a unit test for that:

src/lists/tests/test_views.py (ch14l006)

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post("/lists/new", data={"item_text": ""})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "home.html")
        expected_error = "You can't have an empty list item"
        self.assertContains(response, expected_error)
```

As we're writing this test, we might get slightly offended by the */lists/new* URL, which we're manually entering as a string. We've got a lot of URLs hardcoded in our tests, in our views, and in our templates, which violates the DRY (don't repeat yourself) principle. I don't mind a bit of duplication in tests, but we should definitely be on the lookout for hardcoded URLs in our views and templates, and make a note to refactor them out. But we won't do that straight away, because right now our application is in a broken state. We want to get back to a working state first.

Back to our test, which is failing because the view is currently returning a 302 redirect, rather than a "normal" 200 response:

```
AssertionError: 302 != 200
```

Let's try calling `full_clean()` in the view:

src/lists/views.py (ch14l007)

```
def new_list(request):
    nulist = List.objects.create()
    item = Item.objects.create(text=request.POST["item_text"], list=nulist)
    item.full_clean()
    return redirect(f"/lists/{nulist.id}/")
```

As we're looking at the view code, we find a good candidate for a hardcoded URL to get rid of. Let's add that to our scratchpad:

Now the model validation raises an exception, which comes up through our view:

```
[...]
  File "...goat-book/src/lists/views.py", line 13, in new_list
    item.full_clean()
[...]
django.core.exceptions.ValidationError: {'text': ['This field cannot be
blank.']}
```

So we try our first approach: using a `try/except` to detect errors. Obeying the Testing Goat, we start with just the `try/except` and nothing else. The tests should tell us what to code next.

src/lists/views.py (ch14l010)

```
from django.core.exceptions import ValidationError
[...]

def new_list(request):
    nulist = List.objects.create()
    item = Item.objects.create(text=request.POST["item_text"], list=nulist)
    try:
        item.full_clean()
    except ValidationError:
        pass
    return redirect(f"/lists/{nulist.id}/")
```

That gets us back to the `302 != 200`:

```
AssertionError: 302 != 200
```

Let's return a rendered template then, which should take care of the template check as well:

src/lists/views.py (ch14l011)

```
    except ValidationError:
        return render(request, "home.html")
```

And the tests now tell us to put the error message into the template:

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list
item' in the following response
```

We do that by passing a new template variable in:

src/lists/views.py (ch14l012)

```
    except ValidationError:
        error = "You can't have an empty list item"
        return render(request, "home.html", {"error": error})
```

Hmm, it looks like that didn't quite work:

```
AssertionError: False is not true : Couldn't find 'You can't have an empty list
item' in the following response
```

A little print-based debug…

src/lists/tests/test_views.py (ch14l013)

```
expected_error = "You can't have an empty list item"
print(response.content.decode())
self.assertContains(response, expected_error)
```

…will show us the cause—Django has HTML-escaped the apostrophe:

```
$ python src/manage.py test lists
[...]
                <div class="invalid-feedback">You can&#x27;t have an empty list
item</div>
```

We could hack something like this into our test:

```
        expected_error = "You can&#39;t have an empty list item"
```

But using Django's helper function `html.escape()` is probably a better idea:

src/lists/tests/test_views.py (ch14l014)

```
from django.utils import html
[...]

        expected_error = html.escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

That passes!

```
Ran 13 tests in 0.047s

OK
```

Checking That Invalid Input Isn't Saved to the Database

Before we go further though, did you notice a little logic error we've allowed to creep into our implementation? We're currently creating an object, even if validation fails:

src/lists/views.py

```
    item = Item.objects.create(text=request.POST["item_text"], list=nulist)
    try:
        item.full_clean()
    except ValidationError:
        [...]
```

Let's add a new unit test to make sure that empty list items don't get saved:

src/lists/tests/test_views.py (ch14l015)

```
class NewListTest(TestCase):
    [...]

    def test_validation_errors_are_sent_back_to_home_page_template(self):
        [...]

    def test_invalid_list_items_arent_saved(self):
        self.client.post("/lists/new", data={"item_text": ""})
        self.assertEqual(List.objects.count(), 0)
        self.assertEqual(Item.objects.count(), 0)
```

That gives:

```
[...]
Traceback (most recent call last):
  File "...goat-book/src/lists/tests/test_views.py", line 43, in
test_invalid_list_items_arent_saved
    self.assertEqual(List.objects.count(), 0)
AssertionError: 1 != 0
```

We fix it like this:

src/lists/views.py (ch14l016)

```
def new_list(request):
    nulist = List.objects.create()
    item = Item(text=request.POST["item_text"], list=nulist)
    try:
        item.full_clean()
        item.save()
    except ValidationError:
        nulist.delete()
        error = "You can't have an empty list item"
        return render(request, "home.html", {"error": error})
    return redirect(f"/lists/{nulist.id}/")
```

Do the FTs pass?

```
$ python src/manage.py test functional_tests.test_list_item_validation
[...]
File "...goat-book/src/functional_tests/test_list_item_validation.py", line
32, in test_cannot_add_empty_list_items
    self.wait_for(
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; [...]
```

Not quite, but they did get a little further. Checking the line in which the error occurred (line 31 in my case) we can see that we've got past the first part of the test, and are now onto the second check—that submitting a second empty item also shows an error.

We've got some working code though, so let's have a commit:

```
$ git commit -am "Adjust new list view to do model validation"
```

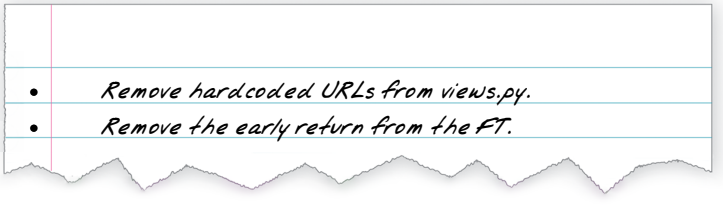
Adding an Early Return to Our FT to Let Us Refactor Against Green

Let's put an early return in the FT to separate what we got working from those that still need to be dealt with:

src/functional_tests/test_list_item_validation.py (ch14l017)

```
class ItemValidationTest(FunctionalTest):
    def test_cannot_add_empty_list_items(self):
        [...]
        self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table("1: Purchase milk")

        return  # TODO re-enable the rest of this test.

        # Perversely, she now decides to submit a second blank list item
        self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)
        [...]
```

We should also remind ourselves not to forget to remove this early return:



And now, we can focus on making our code a little neater.

When working on a new feature, it's common to realise partway through that a refactor of the application is needed. Adding an early return to the FT you're currently working on enables you to perform this refactor against passing FTs, even while the feature is still in progress.

Django Pattern: Processing POST Requests in the Same View That Renders the Form

This time we'll use a slightly different approach—one that's actually a very common pattern in Django—which uses the same view to both process POST requests and render the form that they come from. Whilst this doesn't fit the REST-ful URL model quite as well, it has the important advantage that the same URL can display a form, and display any errors encountered in processing the user's input; see Figure 14-1.

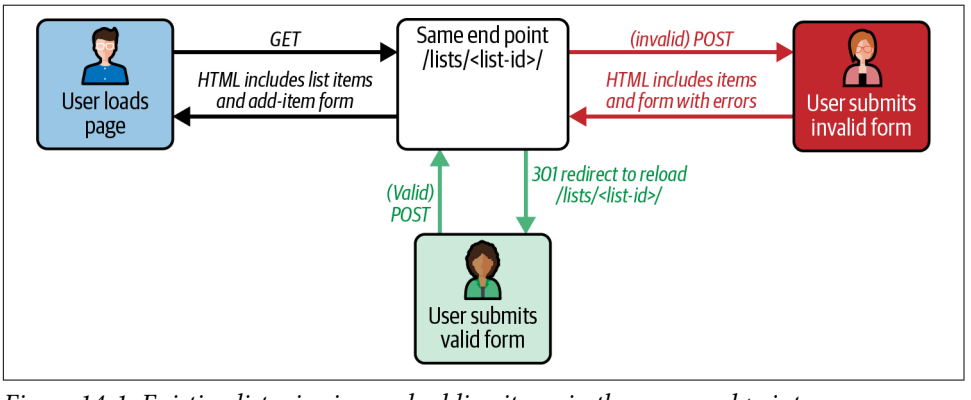


Figure 14-1. Existing list, viewing and adding items in the same end point

The current situation is that we have one view and URL for displaying a list, and one view and URL for processing additions to that list. We're going to combine them into one.

In this section, we're performing a refactor at the application level. We execute our application-level refactor by changing or adding unit tests, and then adjusting our code. We use the functional tests to warn us if we ever go backwards and introduce a regression, and when they're back to green we'll know our refactor is done. Have another look at the diagram from the end of Chapter 4 if you need to get your bearings.

Refactor: Transferring the new_item Functionality into view_list

Let's take the two old tests from `NewItemTest`—the ones that are about saving POST requests to existing lists—and move them into `ListViewTest`. As we do so, we also make them point at the base list URL, instead of *…/add_item*:

src/lists/tests/test_views.py (ch14l030)

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]

    def test_renders_input_form(self):
        mylist = List.objects.create()
        response = self.client.get(f"/lists/{mylist.id}/")
        parsed = lxml.html.fromstring(response.content)
        [form] = parsed.cssselect("form[method=POST]")
        self.assertEqual(form.get("action"), f"/lists/{mylist.id}/")  ❶
        inputs = form.cssselect("input")
        self.assertIn("item_text", [input.get("name") for input in inputs])

    def test_displays_only_items_for_that_list(self):
        [...]

    def test_can_save_a_POST_request_to_an_existing_list(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        self.client.post(
            f"/lists/{correct_list.id}/",  ❷
            data={"item_text": "A new item for an existing list"},
        )

        self.assertEqual(Item.objects.count(), 1)
        new_item = Item.objects.get()
        self.assertEqual(new_item.text, "A new item for an existing list")
        self.assertEqual(new_item.list, correct_list)

    def test_POST_redirects_to_list_view(self):
        other_list = List.objects.create()
        correct_list = List.objects.create()

        response = self.client.post(
            f"/lists/{correct_list.id}/",  ❷
            data={"item_text": "A new item for an existing list"},
        )

        self.assertRedirects(response, f"/lists/{correct_list.id}/")
```

❶ We want our form to point at the base URL.

❷ And the two tests we've merged in need to target the base URL too.

Note that the `NewItemTest` class disappears completely. I've also changed the name of the redirect test to make it explicit that it only applies to POST requests.

That gives:

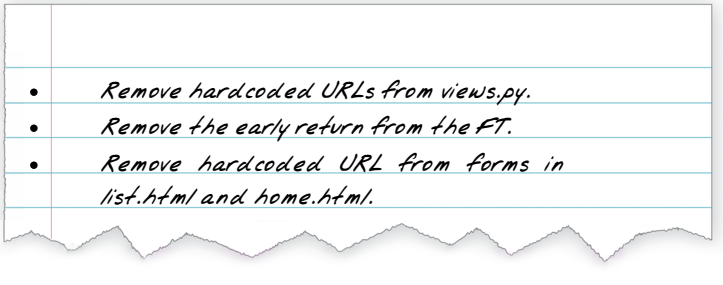
```
FAIL: test_POST_redirects_to_list_view
(lists.tests.test_views.ListViewTest.test_POST_redirects_to_list_view)
[...]
AssertionError: 200 != 302 : Response didn't redirect as expected: Response
code was 200 (expected 302)
[...]
FAIL: test_can_save_a_POST_request_to_an_existing_list (lists.tests.test_views.
ListViewTest.test_can_save_a_POST_request_to_an_existing_list)
[...]
AssertionError: 0 != 1
[...]
FAIL: test_renders_input_form
(lists.tests.test_views.ListViewTest.test_renders_input_form)
[...]
AssertionError: '/lists/1/add_item' != '/lists/1/'
[...]
Ran 14 tests in 0.025s

FAILED (failures=3)
```

That last one is something we can fix in the template. Let's go to *list.html*, and change the `action` attribute on our form so that it points at the existing list URL:

src/lists/templates/list.html (ch14l031)

```
{% block form_action %}/lists/{{ list.id }}/{% endblock %}
```

Incidentally, that's another hardcoded URL. Let's add it to our to-do list and, while we're thinking about it, there's one in *home.html* too:



We're now down to two failing tests:

```
FAIL: test_POST_redirects_to_list_view
(lists.tests.test_views.ListViewTest.test_POST_redirects_to_list_view)
[...]
AssertionError: 200 != 302 : Response didn't redirect as expected: Response
code was 200 (expected 302)
[...]
FAIL: test_can_save_a_POST_request_to_an_existing_list (lists.tests.test_views.
ListViewTest.test_can_save_a_POST_request_to_an_existing_list)
[...]
AssertionError: 0 != 1
[...]
Ran 14 tests in 0.025s

FAILED (failures=2)
```

Those are both about getting the list view to handle POST requests. Let's copy some code across from `add_item` view to do just that:

src/lists/views.py (ch14l032)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    if request.method == "POST":  ❶
        Item.objects.create(text=request.POST["item_text"], list=our_list)  ❷
        return redirect(f"/lists/{our_list.id}/")  ❷
    return render(request, "list.html", {"list": our_list})
```

❶ We add a branch for when the method is POST.

❷ And we copy the `Item.objects.create()` and `redirect()` lines from the `add_item` view.

That gets us passing unit tests!

```
Ran 14 tests in 0.047s

OK
```

Now we can delete the `add_item` view, as it's no longer needed…oops, an unexpected failure:

```
[...]
AttributeError: module 'lists.views' has no attribute 'add_item'
```

It's because we've deleted the view, but it's still being referred to in *urls.py*. We remove it from there:

src/lists/urls.py (ch14l034)

```
urlpatterns = [
    path("new", views.new_list, name="new_list"),
    path("<int:list_id>/", views.view_list, name="view_list"),
]
```

OK, we're back to the green on the unit tests.

```
OK
```

Let's try a full FT run: they're all passing!

```
Ran 4 tests in 9.951s

OK
```

Our refactor of the `add_item` functionality is complete. We should commit there:

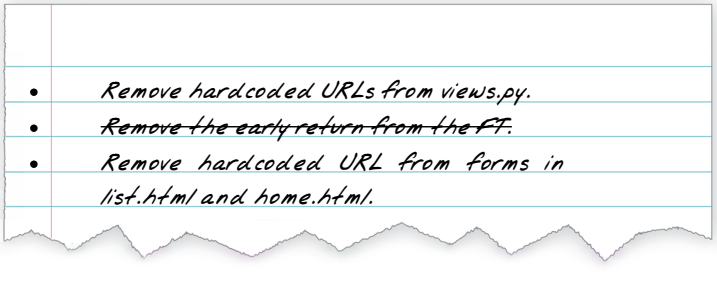
```
$ git commit -am "Refactor list view to handle new item POSTs"
```

We can remove the early return now:

src/functional_tests/test_list_item_validation.py (ch14l035)

```
@@ -24,8 +24,6 @@ class ItemValidationTest(FunctionalTest):
         self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)
         self.wait_for_row_in_list_table("1: Purchase milk")

-        return  # TODO re-enable the rest of this test.
-
         # Perversely, she now decides to submit a second blank list item
```

And, let's cross that off our scratchpad too:



Run the FTs again to see what's still there that needs to be fixed:

```
$ python src/manage.py test functional_tests
[...]
ERROR: test_cannot_add_empty_list_items (functional_tests.test_list_item_valida
tion.ItemValidationTest.test_cannot_add_empty_list_items)
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; [...]

Ran 4 tests in 15.276s
FAILED (errors=1)
```

We're back to working on this one failure in our new FT.

Enforcing Model Validation in view_list

We still want the addition of items to existing lists to be subject to our model validation rules. Let's write a new unit test for that; it's very similar to the one for the home page, with just a couple of tweaks:

src/lists/tests/test_views.py (ch14l036)

```
class ListViewTest(TestCase):
    [...]

    def test_validation_errors_end_up_on_lists_page(self):
        list_ = List.objects.create()
        response = self.client.post(
            f"/lists/{list_.id}/",
            data={"item_text": ""},
        )
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "list.html")
        expected_error = html.escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

Because our view currently does not do any validation, this should fail and just redirect for all POSTs:

```
        self.assertEqual(response.status_code, 200)
AssertionError: 302 != 200
```

Here's an implementation:

src/lists/views.py (ch14l037)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    error = None

    if request.method == "POST":
        try:
            item = Item(text=request.POST["item_text"], list=our_list)  ❶
            item.full_clean()  ❷
            item.save()  ❷
            return redirect(f"/lists/{our_list.id}/")
        except ValidationError:
            error = "You can't have an empty list item"

    return render(request, "list.html", {"list": our_list, "error": error})
```

❶ Notice we do `Item()` instead of `Item.objects.create()`.

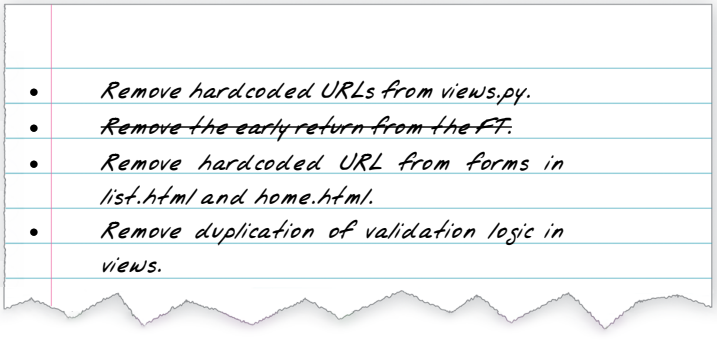
❷ Then we call `full_clean()` before we call `save()`.

It works:

```
Ran 15 tests in 0.047s

OK
```

But it's not deeply satisfying, is it? There's definitely some duplication of code here; that `try/except` occurs twice in *views.py*, and in general things are feeling clunky.

Let's wait a bit before we do more refactoring though, because we know we're about to do some slightly different validation coding for duplicate items. We'll just add it to our scratchpad for now:



One of the reasons that the "three strikes and refactor" rule exists is that, if you wait until you have three use cases, each might be slightly different, and it gives you a better view for what the common functionality is. If you refactor too early, you may find that the third use case doesn't quite fit with your refactored code.

At least our FTs are back to passing:

```
$ python src/manage.py test functional_tests
[...]
OK
```

We're back to a working state, so we can take a look at some of the items on our scratchpad. This would be a good time for a commit (and possibly a tea break):

```
$ git commit -am "enforce model validation in list view"
```

Refactor: Removing Hardcoded URLs

Do you remember those `name=` parameters in *urls.py*? We just copied them across from the default example that Django gave us, and I've been giving them some reasonably descriptive names. Now we find out what they're for:

src/lists/urls.py

```
    path("new", views.new_list, name="new_list"),
    path("<int:list_id>/", views.view_list, name="view_list"),
```

The {% url %} Template Tag

We can replace the hardcoded URL in *home.html* with a Django template tag that refers to the URL's "name":

src/lists/templates/home.html (ch14l038)

```
{% block form_action %}{% url 'new_list' %}{% endblock %}
```

We check that this doesn't break the unit tests:

```
$ python src/manage.py test lists
OK
```

Let's do the other template. This one is more interesting, because we pass it a parameter:

src/lists/templates/list.html (ch14l039)

```
{% block form_action %}{% url 'view_list' list.id %}{% endblock %}
```

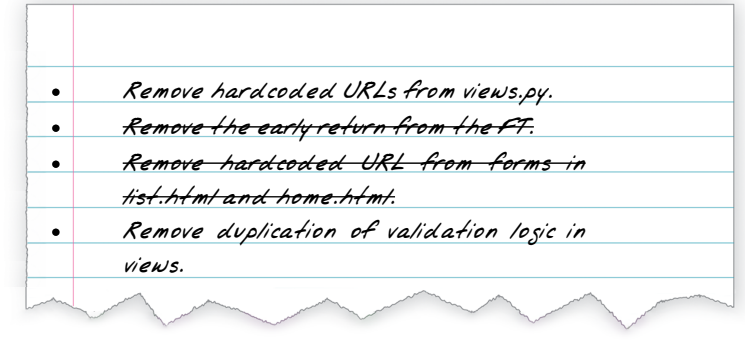
See the Django docs on reverse URL resolution for more info. We run the tests again, and check that they all pass:

```
$ python src/manage.py test lists
OK
$ python src/manage.py test functional_tests
OK
```

Excellent! Let's commit our progress:

```
$ git commit -am "Refactor hard-coded URLs out of templates"
```

And don't forget to cross off the "Remove hardcoded URL…" task as well:



Using get_absolute_url for Redirects

Now let's tackle *views.py*. One way of doing it is just like in the template, passing in the name of the URL and a positional argument:

src/lists/views.py (ch14l040)

```
def new_list(request):
    [...]
    return redirect("view_list", nulist.id)
```

That would get the unit and functional tests passing, but the `redirect` function can do even better magic than that! In Django, because model objects are often associated with a particular URL, you can define a special function called `get_absolute_url` which tells you what page displays the item. It's useful in this case, but it's also useful in the Django admin (which I don't cover in the book, but you'll soon discover for yourself) because it will let you jump from looking at an object in the admin view to looking at the object on the live site. I'd always recommend defining a `get_absolute_url` for a model whenever there is one that makes sense; it takes no time at all.

All it takes is a super simple unit test in *test_models.py*:

src/lists/tests/test_models.py (ch14l041)

```
    def test_get_absolute_url(self):
        mylist = List.objects.create()
        self.assertEqual(mylist.get_absolute_url(), f"/lists/{mylist.id}/")
```

That gives:

```
AttributeError: 'List' object has no attribute 'get_absolute_url'
```

The implementation is to use Django's `reverse` function, which essentially does the reverse of what Django normally does with *urls.py*:

src/lists/models.py (ch14l042)

```
from django.urls import reverse


class List(models.Model):
    def get_absolute_url(self):
        return reverse("view_list", args=[self.id])
```

And now we can use it in the view—the `redirect` function just takes the object we want to redirect to, and it uses `get_absolute_url` under the hood automagically!

src/lists/views.py (ch14l043)

```
def new_list(request):
    [...]
    return redirect(nulist)
```

There's more info in the Django docs. Quick check that the unit tests still pass:

```
OK
```

Then we do the same to `view_list`:

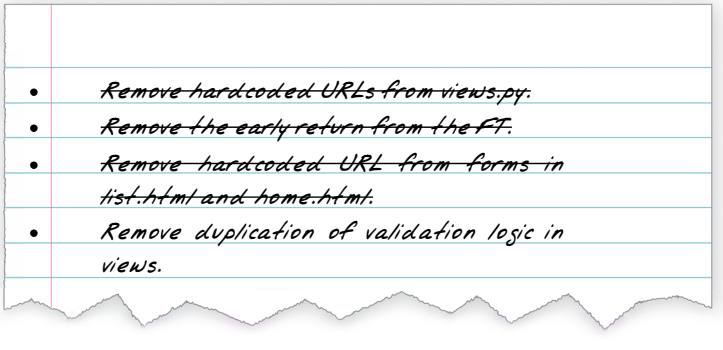
src/lists/views.py (ch14l044)

```
def view_list(request, list_id):
    [...]

            item.save()
            return redirect(our_list)
        except ValidationError:
            error = "You can't have an empty list item"
```

And a full unit test and FT run to assure ourselves that everything still works:

```
$ python src/manage.py test lists
OK
$ python src/manage.py test functional_tests
OK
```

Time to cross off our to-dos…



And commit…

```
$ git commit -am "Use get_absolute_url on List model to DRY urls in views"
```

And we're done with that bit! We have working model-layer validation, and we've taken the opportunity to do a few refactors along the way.

That final scratchpad item will be the subject of the next chapter.

On Database-Layer Validation

As we saw, the specific "not empty" constraint we're trying to apply here isn't enforceable by SQLite, and so it was actually Django that ended up enforcing it for us. However, I always like to push my validation logic down as low as possible:

Validation at the database layer is the ultimate guarantee of data integrity
: It can ensure that, no matter how complex your code gets at the layers above, you have guarantees at the lowest level that your data is valid and consistent.

But it comes at the expense of flexibility
: This benefit doesn't come for free! It's now impossible, even temporarily, to have inconsistent data. Sometimes you might have a good reason for temporarily storing data that breaks the rules rather than storing nothing at all. Perhaps you're importing data from an external source in several stages, for example.

And it's not designed for user-friendliness
: Trying to store invalid data will cause a nasty `IntegrityError` to come back from your database, and possibly the user will see a confusing 500 error page. As we'll see in later chapters, forms-layer validation is designed with the user in mind, anticipating the kinds of helpful error messages we want to send them.

CHAPTER 15

A Simple Form

At the end of the last chapter, we were left with the thought that there was too much duplication in the validation handling bits of our views. Django encourages you to use form classes to do the work of validating user input, and choosing what error messages to display.

We'll use tests to explore the way Django forms work, and then we'll refactor our views to use them. As we go along, we'll see our unit tests and functional tests, in combination, will protect us from regressions.

Moving Validation Logic Into a Form

In Django, a complex view is a code smell. Could some of that logic be pushed out to a form? Or to some custom methods on the model class? Or (perhaps best of all) to a non-Django module that represents your business logic?

Forms have several superpowers in Django:

- They can process user input and validate it for errors.
- They can be used in templates to render HTML input elements, and error messages too.
- And, as we'll see later, some of them can even save data to the database for you.

You don't have to use all three superpowers in every form. You may prefer to roll your own HTML or do your own saving. But they are an excellent place to keep validation logic.

Exploring the Forms API with a Unit Test

Let's do a little experimenting with forms by using a unit test. My plan is to iterate towards a complete solution, and hopefully introduce forms gradually enough that they'll make sense if you've never seen them before.

First we add a new file for our form unit tests, and we start with a test that just looks at the form HTML:

src/lists/tests/test_forms.py (ch15l001)

```
from django.test import TestCase

from lists.forms import ItemForm


class ItemFormTest(TestCase):
    def test_form_renders_item_text_input(self):
        form = ItemForm()
        self.fail(form.as_p())
```

`form.as_p()` renders the form as HTML. This unit test uses a `self.fail` for some exploratory coding. You could just as easily use a `manage.py shell` session, although you'd need to keep reloading your code for each change.

Let's make a minimal form. It inherits from the base `Form` class, and has a single field called `item_text`:

src/lists/forms.py (ch15l002)

```
from django import forms


class ItemForm(forms.Form):
    item_text = forms.CharField()
```

We now see a failure message that tells us what the autogenerated form HTML will look like:

```
AssertionError: <p>
    <label for="id_item_text">Item text:</label>
    <input type="text" name="item_text" required id="id_item_text">
[...]
  </p>
```

It's already pretty close to what we have in *base.html*. We're missing the placeholder attribute and the Bootstrap CSS classes. Let's make our unit test into a test for that:

src/lists/tests/test_forms.py (ch15l003)

```
class ItemFormTest(TestCase):
    def test_form_item_input_has_placeholder_and_css_classes(self):
        form = ItemForm()

        rendered = form.as_p()

        self.assertIn('placeholder="Enter a to-do item"', rendered)
        self.assertIn('class="form-control form-control-lg"', rendered)
```

That gives us a fail, which justifies some real coding:

```
        self.assertIn('placeholder="Enter a to-do item"', rendered)
        ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    AssertionError: 'placeholder="Enter a to-do item"' not found in [...]
```

How can we customise the input for a form field? We use a "widget". Here it is with just the placeholder:

src/lists/forms.py (ch15l004)

```
class ItemForm(forms.Form):
    item_text = forms.CharField(
        widget=forms.widgets.TextInput(
            attrs={
                "placeholder": "Enter a to-do item",
            }
        ),
    )
```

That gives:

```
AssertionError: 'class="form-control form-control-lg"' not found in '<p>\n
<label for="id_item_text">Item text:</label>\n    <input type="text"
name="item_text" placeholder="Enter a to-do item" required id="id_item_text">\n
\n    \n      \n    \n  </p>'
```

And then:

src/lists/forms.py (ch15l005)

```
        widget=forms.widgets.TextInput(
            attrs={
                "placeholder": "Enter a to-do item",
                "class": "form-control form-control-lg",
            }
        ),
```

Doing this sort of `widget` customisation would get tedious if we had a much larger, more complex form. Check out django-crispy-forms for some help.

Development-Driven Tests: Using Unit Tests for Exploratory Coding

Does this feel a bit like development-driven tests? That's OK, now and again.

When you're exploring a new API, you're absolutely allowed to mess about with it for a while before you get back to rigorous TDD. You might use the interactive console, or write some exploratory code (but you have to promise the Testing Goat that you'll throw it away and rewrite it properly later).

Here, we're actually using a unit test as a way of experimenting with the forms API. It can be a surprisingly good way of learning how something works.

Switching to a Django ModelForm

What's next? We want our form to reuse the validation code that we've already defined on our model. Django provides a special class that can autogenerate a form for a model, called `ModelForm`. As you'll see, it's configured using a special inner class called `Meta`:

src/lists/forms.py (ch15l006)

```
from django import forms

from lists.models import Item


class ItemForm(forms.models.ModelForm):
    class Meta:  ❶
        model = Item
        fields = ("text",)

    # item_text = forms.CharField(  ❷
    #     widget=forms.widgets.TextInput(
    #         attrs={
    #             "placeholder": "Enter a to-do item",
    #             "class": "form-control form-control-lg",
    #         }
    #     ),
    # )
```

❶ In `Meta`, we specify which model the form is for and which fields we want it to use.

❷ We'll comment out our manually created field for now.

`ModelForm` does all sorts of smart stuff, like assigning sensible HTML form input types to different types of field, and applying default validation. Check out the docs for more info.

We now have some different-looking form HTML:

```
AssertionError: 'placeholder="Enter a to-do item"' not found in '<p>\n
<label for="id_text">Text:</label>\n    <textarea name="text" cols="40"
rows="10" required id="id_text">\n</textarea>\n    \n    \n      \n    \n
</p>'
```

It's lost our placeholder and CSS class. And you can also see that it's using `name="text"` instead of `name="item_text"`. We can probably live with that. But it's using a `textarea` instead of a normal input, and that's not the UI we want for our app. Thankfully, you can override `widgets` for `ModelForm` fields, similarly to the way we did it with the normal form:

src/lists/forms.py (ch15l007)

```
class ItemForm(forms.models.ModelForm):
    class Meta:
        model = Item
        fields = ("text",)
        widgets = {  ❶
            "text": forms.widgets.TextInput(
                attrs={
                    "placeholder": "Enter a to-do item",
                    "class": "form-control form-control-lg",
                }
            ),
        }
```

❶ We restore some of our commented-out code here, but modified slightly, from being an attribute declaration to a key in a dict.

That gets the test passing.

Testing and Customising Form Validation

Now let's see if the `ModelForm` has picked up the same validation rules that we defined on the model. We'll also learn how to pass data into the form, as if it came from the user:

src/lists/tests/test_forms.py (ch15l008)

```
    def test_form_item_input_has_placeholder_and_css_classes(self):
        [...]

    def test_form_validation_for_blank_items(self):
        form = ItemForm(data={"text": ""})
        form.save()
```

That gives us:

```
ValueError: The Item could not be created because the data didn't validate.
```

Good: the form won't allow you to save if you give it an empty item text. Now let's see if we can get it to use the specific error message that we want. The API for checking form validation *before* we try to save any data is a function called `is_valid`:

src/lists/tests/test_forms.py (ch15l009)

```
def test_form_item_input_has_placeholder_and_css_classes(self):
    [...]

def test_form_validation_for_blank_items(self):
    [...]

def test_form_validation_for_blank_items(self):
    form = ItemForm(data={"text": ""})
    self.assertFalse(form.is_valid())
    self.assertEqual(form.errors["text"], ["You can't have an empty list item"])
```

Calling `form.is_valid()` returns `True` or `False`, but it also has the side effect of validating the input data and populating the `errors` attribute. It's a dictionary mapping the names of fields to lists of errors for those fields (it's possible for a field to have more than one error).

That gives us:

```
AssertionError: ['This field is required.'] != ["You can't have an empty list
item"]
```

Django already has a default error message that we could present to the user—you might use it if you were in a hurry to build your web app, but we care enough to make our message special. Customising it means changing `error_messages`—another `Meta` variable:

src/lists/forms.py (ch15l010)

```
    class Meta:
        model = Item
        fields = ("text",)
        widgets = {
            "text": forms.widgets.TextInput(
                attrs={
                    "placeholder": "Enter a to-do item",
                    "class": "form-control form-control-lg",
                }
            ),
        }
        error_messages = {"text": {"required": "You can't have an empty list item"}}
```

```
OK
```

You know what would be even better than messing about with all these error strings? Having a constant:

src/lists/forms.py (ch15l011)

```
EMPTY_ITEM_ERROR = "You can't have an empty list item"
[...]
        error_messages = {"text": {"required": EMPTY_ITEM_ERROR}}
```

Rerun the tests to see that they pass…OK. Now we can change the tests too.

src/lists/tests/test_forms.py (ch15l012)

```
from lists.forms import EMPTY_ITEM_ERROR, ItemForm
[...]

    def test_form_validation_for_blank_items(self):
        form = ItemForm(data={"text": ""})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors["text"], [EMPTY_ITEM_ERROR])
```

This is a good example of reusing constants in tests. It makes it easier to change the error message later.

And the tests still pass:

```
OK
```

Great. Totes committable:

```
$ git status # should show forms.py and test_forms.py
$ git add src/lists
$ git commit -m "new form for list items"
```

Attempting to Use the Form in Our Views

At this point, we may be tempted to carry on—perhaps extend the form to capture uniqueness validation and empty-item validation.

But there's a sort of corollary to the "deploy as early as possible" Lean methodology, which is "merge code as early as possible". In other words: while building this bit of forms code, it would be easy to go on for ages, adding more and more functionality to the form—I should know, because that's exactly what I did during the drafting of this chapter, and I ended up doing all sorts of work making an all-singing, all-dancing form class before I realised it wouldn't *actually* work for our most basic use case.

So, instead, try to use your new bit of code as soon as possible. This makes sure you never have unused bits of code lying around, and that you start checking your code against "the real world" as soon as possible.

We have a form class that can render some HTML and do validation of at least one kind of error—let's start using it! We should be able to use it in our *base.html* template—so, also, in all of our views.

Using the Form in a View with a GET Request

So, let's start using our form in our home page view:

src/lists/views.py (ch15l013)

```
[...]
from lists.forms import ItemForm
from lists.models import Item, List


def home_page(request):
    return render(request, "home.html", {"form": ItemForm()})
```

OK, now let's try using it in the template—we replace the old `<input ..>` with `{{ form.text }}`:

src/lists/templates/base.html (ch15l014)

```
<form method="POST" action="{% block form_action %}{% endblock %}" >
  {{ form.text }}  ❶
  {% csrf_token %}
  {% if error %}
    <div class="invalid-feedback">{{ error }}</div>
  {% endif %}
</form>
```

❶ `{{ form.text }}` renders just the HTML input for the `text` field of the form.

That causes our two unit tests that check on the form input to fail:

```
[...]
======================================================================
FAIL: test_renders_input_form
(lists.tests.test_views.HomePageTest.test_renders_input_form)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/lists/tests/test_views.py", line 19, in
test_renders_input_form
    self.assertIn("item_text", [input.get("name") for input in inputs])
    ~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'item_text' not found in ['text', 'csrfmiddlewaretoken']  ❶

======================================================================
FAIL: test_renders_input_form
(lists.tests.test_views.ListViewTest.test_renders_input_form)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/lists/tests/test_views.py", line 60, in
test_renders_input_form
    self.assertIn("item_text", [input.get("name") for input in inputs])
    ~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'item_text' not found in ['csrfmiddlewaretoken']  ❷

Ran 18 tests in 0.022s

FAILED (failures=2)
```

❶ The test for the home page is failing because the `name` attribute of the input box is now `text`, not `item_text`.

❷ The test for the list view is failing because because we're not instantiating a form in that view, so there's no `form` variable in the template. The input box isn't even being rendered.

Let's fix things one at a time. First, let's back out our change and restore the hand-crafted HTML input in cases where `{{ form }}` is not defined:

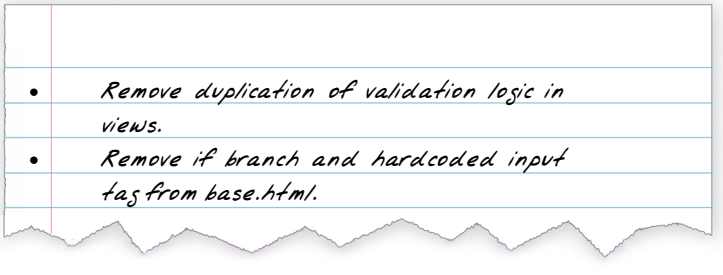
src/lists/templates/base.html (ch15l015)

```
<form method="POST" action="{% block form_action %}{% endblock %}" >
  {% if form %}
    {{ form.text }}
  {% else %}
    <input
      class="form-control form-control-lg {% if error %}is-invalid{% endif %}"
      name="item_text"
      id="id_new_item"
      placeholder="Enter a to-do item"
    />
  {% endif %}
  {% csrf_token %}
  {% if error %}
    <div class="invalid-feedback">{{ error }}</div>
  {% endif %}
</form>
```

That takes us down to one failure:

```
AssertionError: 'item_text' not found in ['text', 'csrfmiddlewaretoken']
```

Let's make a note to come back and tidy this up, and then we'll talk about what's happened and how to deal with it:



The Trade-offs of Django ModelForms: The Frontend Is Coupled to the Database

This highlights one of the trade-offs of using `ModelForm`: by auto-generating the form from the model, we tie the `name=` attribute of our form's HTML `<input>` to the name of the model field in the database.

In a simple CRUD (create, read, update, and delete) app like ours, that's probably a good deal. But it does mean we need to go back and change our assumptions about what the `name=` attribute of the input box is going to be.

While we're at it, it's worth doing an FT run too:

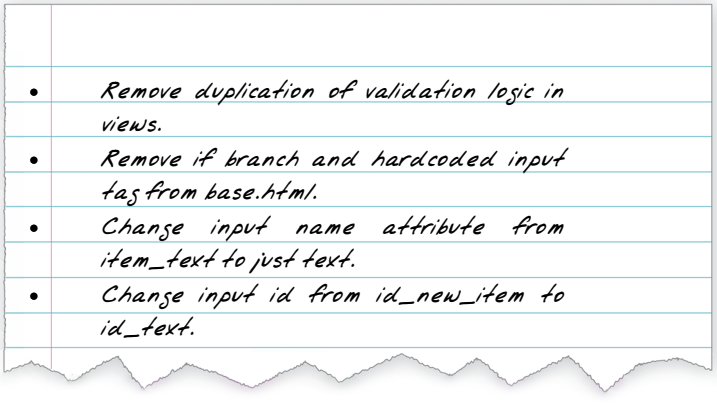
```
$ python src/manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: [id="id_new_item"]; [...]
[...]

FAILED (errors=4)
```

Looks like something else has changed.

If you pause the FTs or inspect the HTML manually in a browser, you'll see that the `ModelForm` also changes the `id` attribute to being `id_text`.[1]

A Big Find-and-Replace

If we want to change our assumption about these two attributes, we'll need to embark on a couple of big find-and-replaces basically:



But before we do that, let's back out the rest of our changes and get back to a working state.

1 It's actually possible to customise this attribute via the `widgets` attribute we used earlier, even on a `ModelForm`, but because you cannot change the `name` one, we may as well just accept this too.

Backing Out Our Changes and Getting to a Working State

The simplest way to back out changes is with `git`. But in this case, leaving a couple of placeholders does no harm, and they'll be helpful to come back to later.

So we can leave the `{{ form.text }}` in the HTML but, by backing out the change in the view, we'll make sure that branch is never actually exercised. Again, to leave ourselves a little placeholder, we'll comment out our code rather than deleting it:

src/lists/views.py (ch15l016)

```
def home_page(request):
    # return render(request, "home.html", {"form": ItemForm()})
    return render(request, "home.html")
```

Be very cautious about leaving commented-out code and unused `if` branches lying around. Do so only if you're sure you're coming back to them very soon, otherwise your codebase will soon get messy!

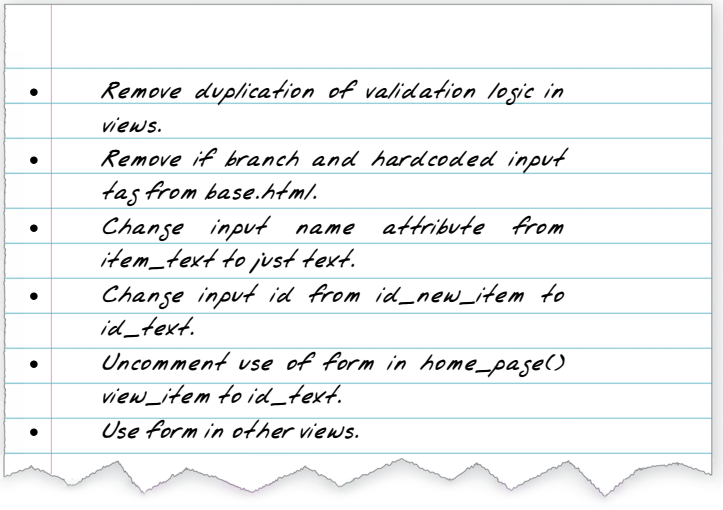
Now we can do a full unit test and FT run to confirm we're back to a working state:

```
$ python src/manage.py test lists
Found 18 test(s).
[...]
OK

$ python src/manage.py test functional_tests
Found 4 test(s).
[...]

OK
```

And let's do a commit to be able to separate out the rename from anything else:

```
$ git diff # changes in base.html + views.py
$ git commit -am "Placeholders for using form in view+template, not in use yet"
```

And pop an item on the to-do list:



Renaming the name Attribute

So, let's have a look for `item_text` in the codebase:

```
$ grep -Ir item_text src
src/lists/migrations/0003_list.py:        ("lists", "0002_item_text"),
src/lists/tests/test_views.py:        self.assertIn("item_text",
[input.get("name") for input in inputs])
[...]
src/lists/templates/base.html:                name="item_text"
src/lists/views.py:    item = Item(text=request.POST["item_text"], list=nulist)
src/lists/views.py:            item = Item(text=request.POST["item_text"],
list=our_list)
```

We can ignore the migration, which is just using `item_text` as metadata. So the changes we need to make are in three places:

1. *views.py*
2. *test_views.py*
3. *base.html*

Let's go ahead and make those. I'm sure you can manage your own find-and-replace! They should look something like this:

src/lists/tests/test_views.py (ch15l017)

```
@@ -16,12 +16,12 @@ class HomePageTest(TestCase):
         [form] = parsed.cssselect("form[method=POST]")
         self.assertEqual(form.get("action"), "/lists/new")
         inputs = form.cssselect("input")
-        self.assertIn("item_text", [input.get("name") for input in inputs])
+        self.assertIn("text", [input.get("name") for input in inputs])


 class NewListTest(TestCase):
     def test_can_save_a_POST_request(self):
-        self.client.post("/lists/new", data={"item_text": "A new list item"})
+        self.client.post("/lists/new", data={"text": "A new list item"})
         self.assertEqual(Item.objects.count(), 1)
         new_item = Item.objects.get()
         self.assertEqual(new_item.text, "A new list item")
[...]
```

Or, in *views.py*:

src/lists/views.py (ch15l019)

```
@@ -12,7 +12,7 @@ def home_page(request):

 def new_list(request):
     nulist = List.objects.create()
-    item = Item(text=request.POST["item_text"], list=nulist)
+    item = Item(text=request.POST["text"], list=nulist)
     try:
         item.full_clean()
         item.save()
@@ -29,7 +29,7 @@ def view_list(request, list_id):

     if request.method == "POST":
         try:
-            item = Item(text=request.POST["item_text"], list=our_list)
+            item = Item(text=request.POST["text"], list=our_list)
             item.full_clean()
             item.save()
             return redirect(our_list)
```

Finally, in *base.html*:

src/lists/templates/base.html (ch15l020)

```
@@ -21,7 +21,7 @@
             {% else %}
               <input
                 class="form-control form-control-lg {% if error %}is-invalid{% endif %}"
-                name="item_text"
+                name="text"
                 id="id_new_item"
                 placeholder="Enter a to-do item"
               />
```

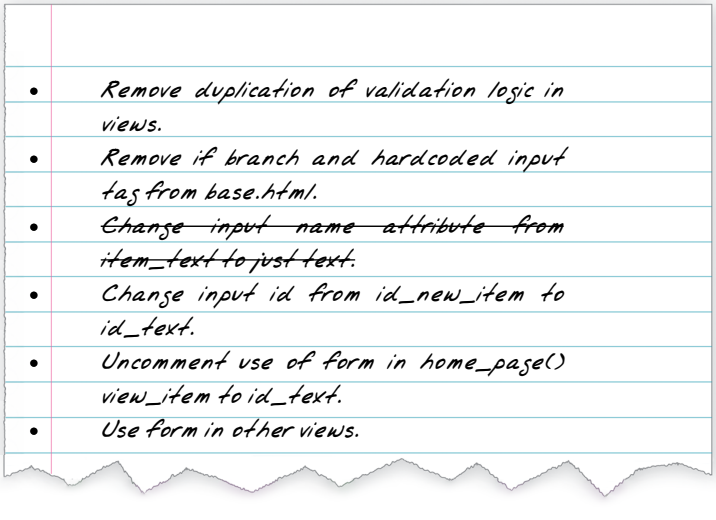
Once you're done, rerun the unit tests to confirm that the application is self-consistent:

```
$ python src/manage.py test lists
[...]
Ran 18 tests in 0.126s

OK
```

And rerun the FTs too:

```
$ python src/manage.py test functional_tests
[...]
Ran 4 tests in 12.154s

OK
```

Good! One down:



Renaming the id Attribute

Now for the `id=` attribute. A quick `grep` shows us that `id_new_item` appears in the template, and in all three FT files:

```
$ grep -r id_new_item
src/lists/templates/base.html:                id="id_new_item"
src/functional_tests/test_list_item_validation.py:
self.browser.find_element(By.ID, "id_new_item").send_keys(Keys.ENTER)
src/functional_tests/test_list_item_validation.py:
self.browser.find_element(By.ID, "id_new_item").send_keys("Purchase milk")
[...]
```

That's a good call for a refactor within the FTs too. Let's make a new helper method in *base.py*:

src/functional_tests/base.py (ch15l021)

```
class FunctionalTest(StaticLiveServerTestCase):
    [...]
    def get_item_input_box(self):
        return self.browser.find_element(By.ID, "id_new_item")  ❶
```

❶ We'll keep the old `id` for now. Working state to working state!

And then we use it throughout—I had to make four changes in *test_simple_list_creation.py*, two in *test_layout_and_styling.py*, and six in *test_list_item_validation.py*, for example:

src/functional_tests/test_simple_list_creation.py

```
        # She is invited to enter a to-do item straight away
        inputbox = self.get_item_input_box()
```

Or:

src/functional_tests/test_list_item_validation.py

```
        # an empty list item. She hits Enter on the empty input box
        self.browser.get(self.live_server_url)
        self.get_item_input_box().send_keys(Keys.ENTER)
```

I won't show you every single one; I'm sure you can manage this for yourself! You can redo the `grep` to check that you've caught them all:

```
$ grep -r id_new_item
src/lists/templates/base.html:                id="id_new_item"
src/functional_tests/base.py:        return self.browser.find_element(By.ID,
"id_new_item")
```

And we can do an FT run too, to make sure we haven't broken anything:

```
$ python src/manage.py test functional_tests
[...]
Ran 4 tests in 12.154s

OK
```

Good! FT refactor complete—now hopefully we can make the application-level refactor of the `id` attribute in just two places, and we've been in a working state the whole way through.

In the FT helper method:

src/functional_tests/base.py (ch15l023)

```
@@ -43,4 +43,4 @@ class FunctionalTest(StaticLiveServerTestCase):
                 time.sleep(0.5)

     def get_item_input_box(self):
-        return self.browser.find_element(By.ID, "id_new_item")
+        return self.browser.find_element(By.ID, "id_text")
```

And in the template:

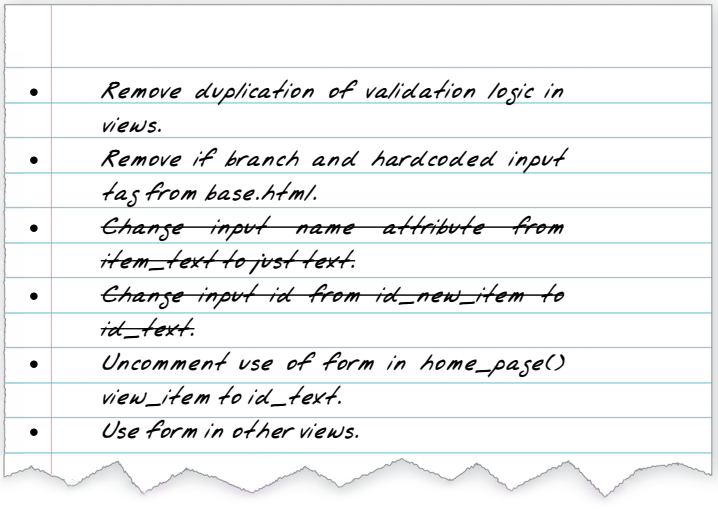
src/lists/templates/base.html (ch15l024)

```
@@ -22,7 +22,7 @@
               <input
                 class="form-control form-control-lg {% if error %}is-invalid{% endif %}"
                 name="text"
-                id="id_new_item"
+                id="id_text"
                 placeholder="Enter a to-do item"
               />
             {% endif %}
```

And an FT run to confirm:

```
$ python src/manage.py test functional_tests
[...]
Ran 4 tests in 12.154s

OK
```

Hooray!



A Second Attempt at Using the Form in Our Views

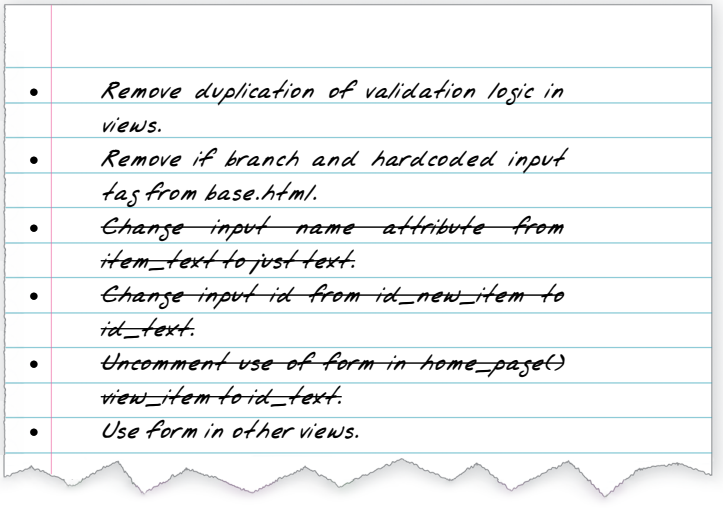
Now that we've done the groundwork, hopefully we can drop in our form in the `home_page()` once again:

src/lists/views.py (ch15l025)

```
def home_page(request):
    return render(request, "home.html", {"form": ItemForm()})
```

Looking good!

```
$ python src/manage.py test lists
Found 18 test(s).
[...]
OK
```



Let's see what happens if we remove that `if` from the template:

src/lists/templates/base.html (ch15l026)

```
@@ -16,16 +16,7 @@
           <h1 class="display-1 mb-4">{% block header_text %}{% endblock %}</h1>

           <form method="POST" action="{% block form_action %}{% endblock %}" >
-            {% if form %}
-              {{ form.text }}
-            {% else %}
-              <input
-                class="form-control form-control-lg {% if error %}is-invalid{% endif %}"
-                name="text"
-                id="id_text"
-                placeholder="Enter a to-do item"
-              />
-            {% endif %}
+            {{ form.text }}
             {% csrf_token %}
             {% if error %}
               <div class="invalid-feedback">{{ error }}</div>
```

Aha—the unit tests are there to tell us that we need to use the form in `view_list()` too:

```
AssertionError: 'text' not found in ['csrfmiddlewaretoken']
```

Here's the minimal use of the form—we won't use it for validation yet, just for getting the form into the template:

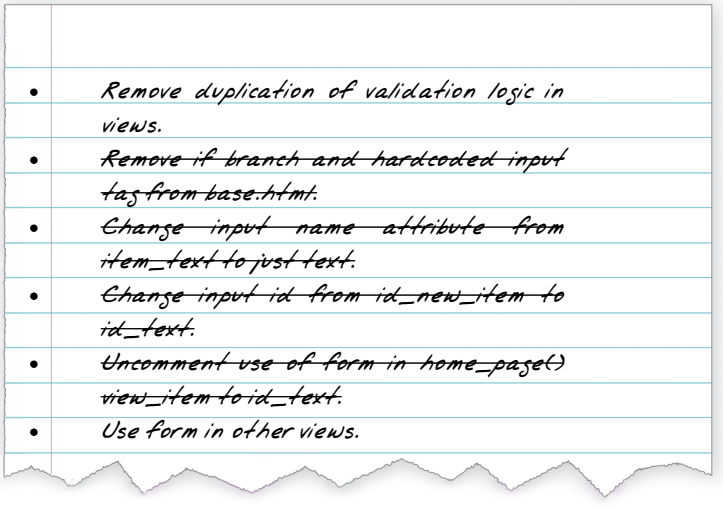
src/lists/views.py (ch15l027)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    error = None
    form = ItemForm()

    if request.method == "POST":
        try:
            item = Item(text=request.POST["text"], list=our_list)
            item.full_clean()
            item.save()
            return redirect(our_list)
        except ValidationError:
            error = "You can't have an empty list item"

    return render(
        request, "list.html", {"list": our_list, "form": form, "error": error}
    )
```

And the tests are happy with that too:

```
$ python src/manage.py test lists
Found 18 test(s).
[...]
OK
```

We're done with the template; what's next?



Right, let's move on to the next view that doesn't use our form yet—`new_list()`. And actually, that'll help us with the first item, which was the whole point of this adventure, really: to see if the forms can help us better handle validation.

Let's see how that works now.

Using the Form in a View That Takes POST Requests

Here's how we can use the form in the `new_list()` view, avoiding all the manual manipulation of `request.POST` and the error message:

src/lists/views.py (ch15l028)

```
def new_list(request):
    form = ItemForm(data=request.POST)  ❶
    if form.is_valid():  ❷
        nulist = List.objects.create()
        Item.objects.create(text=request.POST["text"], list=nulist)
        return redirect(nulist)
    else:
        return render(request, "home.html", {"form": form})  ❸
```

❶ We pass the `request.POST` data into the form's constructor.

❷ We use `form.is_valid()` to determine whether this is a good or a bad submission.

❸ In the invalid case, we pass the form down to the template, instead of our hardcoded error string.

That view is now looking much nicer!

But, we have a regression in the unit tests:

```
======================================================================
FAIL: test_validation_errors_are_sent_back_to_home_page_template (lists.tests.t
est_views.NewListTest.test_validation_errors_are_sent_back_to_home_page_templat
e)
 ----------------------------------------------------------------------
[...]
    self.assertContains(response, expected_error)
    ~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: False is not true : Couldn't find 'You can&#x27;t have an empty
list item' in the following response
b'<!doctype html>\n<html lang="en">\n\n  <head>\n    <title>To-Do
[...]
```

Using the Form to Display Errors in the Template

We're failing because we're not yet *using* the form to display errors in the template. Here's how to do that:

src/lists/templates/base.html (ch15l029)

```
<form method="POST" action="{% block form_action %}{% endblock %}" >
  {{ form.text }}
  {% csrf_token %}
  {% if form.errors %}  ❶
    <div class="invalid-feedback">{{ form.errors.text }}</div>  ❷
  {% endif %}
</form>
```

❶ We change the `if` to look at `form.errors`: it contains a list of all the errors for the form.

❷ `form.errors.text` is magical Django template syntax for `form.errors["text"]`—i.e., the list of errors for the text field in particular.

What does that do to our unit tests?

```
======================================================================
FAIL: test_validation_errors_end_up_on_lists_page (lists.tests.test_views.ListV
iewTest.test_validation_errors_end_up_on_lists_page)
 ----------------------------------------------------------------------
[...]
AssertionError: False is not true : Couldn't find 'You can&#x27;t have an empty
list item' in the following response
```

An unexpected failure—it's actually in the tests for our final view, `view_list()`. Once again, because we've changed the base template, which is used by *all* views, we've made a change that impacts more places than we intended. Let's follow our standard pattern, get back to a working state, and see if we can dig into this a bit.

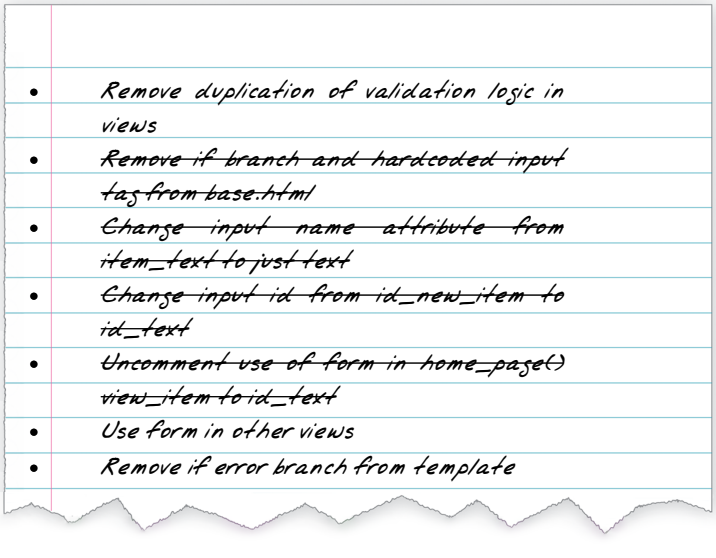
Get Back to a Working State

Let's restore the old `[% if %]` in the template, so we display errors in both old and new cases:

src/lists/templates/base.html (ch15l029-1)

```
<form method="POST" action="{% block form_action %}{% endblock %}" >
  {{ form.text }}
  {% csrf_token %}
  {% if error %}
    <div class="invalid-feedback">{{ error }}</div>
  {% endif %}
  {% if form.errors %}
    <div class="invalid-feedback">{{ form.errors.text }}</div>
  {% endif %}
</form>
```

And add an item to our stack:



A Helper Method for Several Short Tests

Let's take a look at our tests for both views, particularly the ones that check for invalid inputs:

src/lists/tests/test_views.py

```
class NewListTest(TestCase):
    [...]
    def test_validation_errors_are_sent_back_to_home_page_template(self):
        response = self.client.post("/lists/new", data={"text": ""})
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "home.html")
        expected_error = html.escape("You can't have an empty list item")
        self.assertContains(response, expected_error)

    def test_invalid_list_items_arent_saved(self):
        self.client.post("/lists/new", data={"text": ""})
        self.assertEqual(List.objects.count(), 0)
        self.assertEqual(Item.objects.count(), 0)

class ListViewTest(TestCase):
    [...]
    def test_validation_errors_end_up_on_lists_page(self):
        list_ = List.objects.create()
        response = self.client.post(
            f"/lists/{list_.id}/",
            data={"text": ""},
        )
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "list.html")
        expected_error = html.escape("You can't have an empty list item")
        self.assertContains(response, expected_error)
```

I see a few problems here:

1. We're explicitly checking that validation errors prevent anything from being saved to the database in `NewListTest`, but not in `ListViewTest`.
2. We're mixing up the test for the status code, the template, and finding the error in the result.

Let's be extra meticulous here, and separate out these concerns. Ideally, each test should have one assert. If we used copy-paste, that would start to involve a lot of duplication, so using a couple of helper methods is a good idea here.

Here's some better tests in `NewListTest`:

src/lists/tests/test_views.py (ch15l029-2)

```
from lists.forms import EMPTY_ITEM_ERROR
[...]

class NewListTest(TestCase):
    def test_can_save_a_POST_request(self):
        [...]
    def test_redirects_after_POST(self):
        [...]

    def post_invalid_input(self):
        return self.client.post("/lists/new", data={"text": ""})

    def test_for_invalid_input_nothing_saved_to_db(self):
        self.post_invalid_input()
        self.assertEqual(Item.objects.count(), 0)

    def test_for_invalid_input_renders_list_template(self):
        response = self.post_invalid_input()
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "home.html")

    def test_for_invalid_input_shows_error_on_page(self):
        response = self.post_invalid_input()
        self.assertContains(response, html.escape(EMPTY_ITEM_ERROR))
```

By making a little helper function, `post_invalid_input()`, we can make three separate tests without duplicating lots of lines of code. We've seen this several times now. It often feels more natural to write view tests as a single, monolithic block of assertions—the view should do this and this and this, then return that with this.

But breaking things out into multiple tests is often worthwhile; as we saw in previous chapters, it helps you isolate the exact problem you have when you later accidentally introduce a bug. Helper methods are one of the tools that lower the psychological barrier, by reducing boilerplate and keeping the tests readable.

Let's do something similar in `ListViewTest`:

src/lists/tests/test_views.py (ch15l029-3)

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        [...]
    def test_renders_input_form(self):
        [...]
    def test_displays_only_items_for_that_list(self):
        [...]
    def test_can_save_a_POST_request_to_an_existing_list(self):
        [...]
    def test_POST_redirects_to_list_view(self):
        [...]

    def post_invalid_input(self):
        mylist = List.objects.create()
        return self.client.post(f"/lists/{mylist.id}/", data={"text": ""})

    def test_for_invalid_input_nothing_saved_to_db(self):
        self.post_invalid_input()
        self.assertEqual(Item.objects.count(), 0)

    def test_for_invalid_input_renders_list_template(self):
        response = self.post_invalid_input()
        self.assertEqual(response.status_code, 200)
        self.assertTemplateUsed(response, "list.html")

    def test_for_invalid_input_shows_error_on_page(self):
        response = self.post_invalid_input()
        self.assertContains(response, html.escape(EMPTY_ITEM_ERROR))
```

And let's rerun all our tests:

```
$ python src/manage.py test lists
Found 21 test(s).
[...]
OK
```

Great! We now feel confident that we have a lot of very specific unit tests, which can point us to exactly what goes wrong if we ever make a mistake.

So let's have another go at using our form for *all* views, by fully committing to the `{{ form.errors }}` in the template:

src/lists/templates/base.html (ch15l029-4)

```
@@ -18,9 +18,6 @@
           <form method="POST" action="{% block form_action %}{% endblock %}" >
             {{ form.text }}
             {% csrf_token %}
-            {% if error %}
-              <div class="invalid-feedback">{{ error }}</div>
-            {% endif %}
             {% if form.errors %}
               <div class="invalid-feedback">{{ form.errors.text }}</div>
             {% endif %}
```

And we'll see that exactly one test is failing:

```
FAIL: test_for_invalid_input_shows_error_on_page (lists.tests.test_views.ListVi
ewTest.test_for_invalid_input_shows_error_on_page)
[...]
AssertionError: False is not true : Couldn't find 'You can&#x27;t have an empty
list item' in the following response
```

Using the Form in the Existing Lists View

Let's try and work step by step towards fully using our form in this final view.

Using the Form to Pass Errors to the Template

At the moment, one test is failing because the `view_list()` view for existing lists is not populating `form.errors` in the invalid case. Let's address just that:

src/lists/views.py (ch15l030-1)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    error = None
    form = ItemForm()  ❷

    if request.method == "POST":
        form = ItemForm(data=request.POST)  ❶
        try:
            item = Item(text=request.POST["text"], list=our_list)
            item.full_clean()
            item.save()
            return redirect(our_list)
        except ValidationError:
            error = "You can't have an empty list item"

    return render(
        request, "list.html", {"list": our_list, "form": form, "error": error}  ❸
    )
```

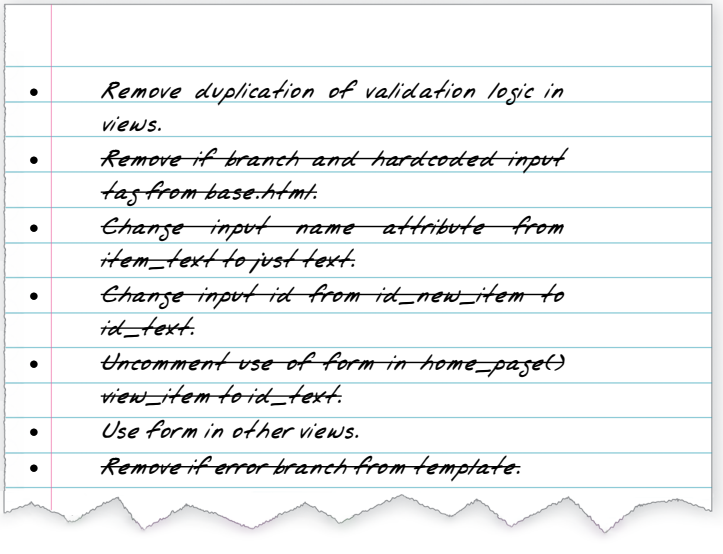
❶ Let's add this line, in the `method=POST` branch, and instantiate a form using the POST data.

❷ We already had this empty form for the GET case, but our new one will override it.

❸ And it should now drop through to the template here.

That gets us back to a working state!

```
Found 21 test(s).
[...]
OK
```



Refactoring the View to Use the Form Fully

Now let's start using the form more fully, and remove some of the manual error handling.

We remove the `try/except` and replace it with an `if form.is_valid()` check, like the one in `new_list()`:

src/lists/views.py (ch15l030-2)

```
@@ -26,13 +26,11 @@ def view_list(request, list_id):

     if request.method == "POST":
         form = ItemForm(data=request.POST)
-        try:
+        if form.is_valid():
             item = Item(text=request.POST["text"], list=our_list)
             item.full_clean()
             item.save()
             return redirect(our_list)
-        except ValidationError:
-            error = "You can't have an empty list item"
     return render(
         request, "list.html", {"list": our_list, "form": form, "error": error}
```

And the tests still pass:

```
OK
```

Next, we no longer need the `.full_clean()`, so we can go back to using `.objects.create()`:

src/lists/views.py (ch15l030-3)

```
@@ -27,9 +27,7 @@ def view_list(request, list_id):
     if request.method == "POST":
         form = ItemForm(data=request.POST)
         if form.is_valid():
-            item = Item(text=request.POST["text"], list=our_list)
-            item.full_clean()
-            item.save()
+            Item.objects.create(text=request.POST["text"], list=our_list)
             return redirect(our_list)
```

The tests still pass:

```
OK
```

Finally, the `error` variable is always `None`, and is no longer needed in the template anyhow:

src/lists/views.py (ch15l030-4)

```
@@ -21,7 +21,6 @@ def new_list(request):

 def view_list(request, list_id):
     our_list = List.objects.get(id=list_id)
-    error = None
     form = ItemForm()

     if request.method == "POST":
@@ -30,6 +29,4 @@ def view_list(request, list_id):
             Item.objects.create(text=request.POST["text"], list=our_list)
             return redirect(our_list)

-    return render(
-        request, "list.html", {"list": our_list, "form": form, "error": error}
-    )
+    return render(request, "list.html", {"list": our_list, "form": form})
```

And the tests are happy with that!

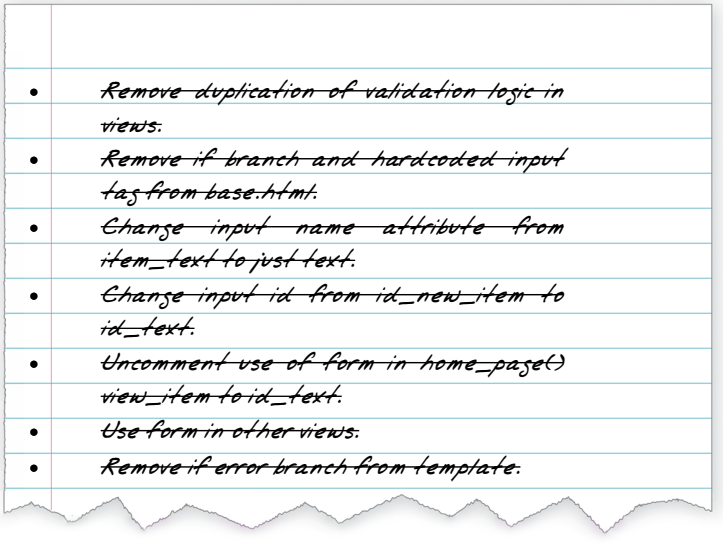
```
OK
```

I think our view is in a pretty good shape now. Here it is in non-diff mode, as a recap:

src/lists/views.py

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    form = ItemForm()

    if request.method == "POST":
        form = ItemForm(data=request.POST)
        if form.is_valid():
            Item.objects.create(text=request.POST["text"], list=our_list)
            return redirect(our_list)

    return render(request, "list.html", {"list": our_list, "form": form})
```

I think we can give ourselves the satisfaction of doing some crossing-things-out:



Phew!

Hey, it's been a while, what do our FTs think?

```
[...]
======================================================================
ERROR: test_cannot_add_empty_list_items (functional_tests.test_list_item_valida
tion.ItemValidationTest.test_cannot_add_empty_list_items)
 ----------------------------------------------------------------------
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; [...]
[...]

Ran 4 tests in 14.897s

FAILED (errors=1)
```

Oh. All the regression tests are OK, but our validation test seems to be failing—and failing early too! It's on the first attempt to submit an empty item. What happened?

An Unexpected Benefit: Free Client-Side Validation from HTML5

How shall we find out what's going on here? One option is to add the usual `time.sleep` just before the error in the FTs, and take a look at what's happening while they run. Alternatively, spin up the site manually with `manage.py runserver` if you prefer. Either way, you should see something like Figure 15-1.

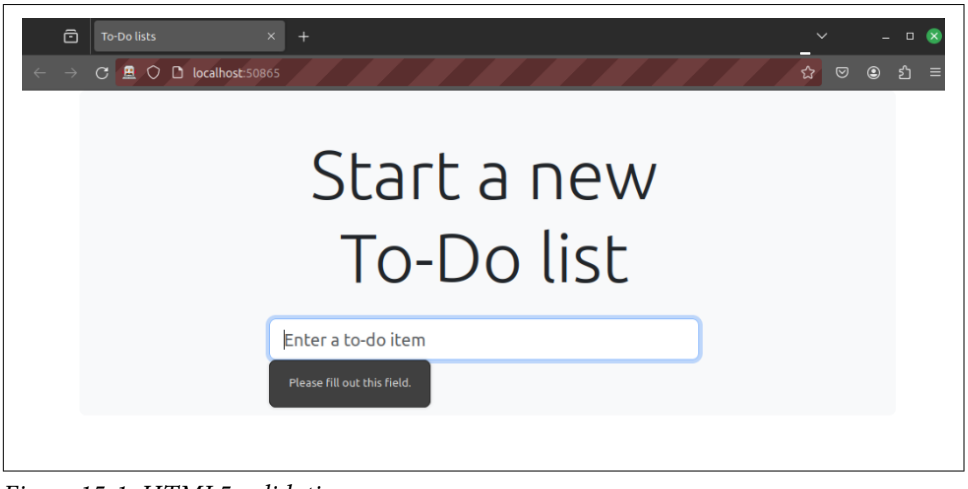


Figure 15-1. HTML5 validation says no

It seems like the browser is preventing the user from even submitting the input when it's empty. It's because Django has added the `required` attribute to the HTML input (take another look at our `as_p()` printouts from earlier if you don't believe me, or have a look at the source in DevTools).

This is a feature of HTML5; browsers nowadays will do some validation at the client side if they can, preventing users from even submitting invalid input. That's actually good news!

But, we were working based on incorrect assumptions about what the user experience was going to be. Let's change our FT to reflect this new expectation:

src/functional_tests/test_list_item_validation.py (ch15l031)

```
class ItemValidationTest(FunctionalTest):
    def test_cannot_add_empty_list_items(self):
        # Edith goes to the home page and accidentally tries to submit
        # an empty list item. She hits Enter on the empty input box
        self.browser.get(self.live_server_url)
        self.get_item_input_box().send_keys(Keys.ENTER)

        # The browser intercepts the request, and does not load the list page
        self.wait_for(
            lambda: self.browser.find_element(By.CSS_SELECTOR, "#id_text:invalid")  ❶
        )

        # She starts typing some text for the new item and the error disappears
        self.get_item_input_box().send_keys("Purchase milk")
        self.wait_for(
            lambda: self.browser.find_element(By.CSS_SELECTOR, "#id_text:valid")  ❷
        )

        # And she can submit it successfully
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table("1: Purchase milk")

        # Perversely, she now decides to submit a second blank list item
        self.get_item_input_box().send_keys(Keys.ENTER)

        # Again, the browser will not comply
        self.wait_for_row_in_list_table("1: Purchase milk")
        self.wait_for(
            lambda: self.browser.find_element(By.CSS_SELECTOR, "#id_text:invalid")
        )

        # And she can make it happy by filling some text in
        self.get_item_input_box().send_keys("Make tea")
        self.wait_for(
            lambda: self.browser.find_element(
                By.CSS_SELECTOR,
                "#id_text:valid",
            )
        )
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table("2: Make tea")
```

❶ Instead of checking for our custom error message, we check using the CSS pseudo-selector `:invalid`, which the browser applies to any HTML5 input that has invalid input.

❷ And we check for its converse in the case of valid inputs.

See how useful and flexible our `self.wait_for()` function is turning out to be?

Our FT does look quite different from how it started though, doesn't it? I'm sure that's raising a lot of questions in your mind right now. Put a pin in them for a moment; I promise we'll talk. Let's first see if we're back to passing tests:

```
$ python src/manage.py test functional_tests
[...]
Ran 4 tests in 12.154s

OK
```

A Pat on the Back

First, let's give ourselves a massive pat on the back: we've just made a major change to our small app—that input field, with its name and ID, is absolutely critical to making everything work. We've touched seven or eight different files, doing a refactor that's quite involved…this is the kind of thing that, without tests, would seriously worry me. In fact, I might well have decided that it wasn't worth messing with code that works. But, because we have a full test suite, we can delve around, tidying things up, safe in the knowledge that the tests are there to spot any mistakes we make. It just makes it that much more likely that you're going to keep refactoring, keep tidying up, keep gardening, keep tending to your code, and keep everything neat and tidy and clean and smooth and precise and concise and functional and good.

And it's definitely time for a commit:

```
$ git diff
$ git commit -am "use form in all views, back to working state"
```

But Have We Wasted a Lot of Time?

But what about our custom error message? What about all that effort rendering the form in our HTML template? We're not even passing those errors from Django to the user if the browser is intercepting the requests before the user even makes them! And our FT isn't even testing that stuff any more!

Well, you're quite right. But there are two or three reasons all our time hasn't been wasted. Firstly, client-side validation isn't enough to guarantee you're protected from bad inputs, so you always need the server side as well if you really care about data integrity; using a form is a nice way of encapsulating that logic.

Also, not all browsers fully implement HTML5,[2] so some users might still see our custom error message. And if or when we come to letting users access our data via an API (see Online Appendix: Building a REST API), then our validation messages will come back into use. On top of that, we'll be able to reuse all our validation and forms

2 Safari was a notable laggard in the last decade; it's up to date now.

code when we do some more advanced validation that can't be done by HTML5 magic.

But you know, even if all that weren't true, you can't be too hard on yourself for occasionally barking up the wrong tree while you're coding. None of us can see the future, and we should concentrate on finding the right solution rather than the time "wasted" on the wrong solution.

Using the ModelForm's Own Save Method

There are a couple more things we can do to make our views even simpler. I've mentioned that forms are supposed to be able to save data to the database for us. Our case won't quite work out of the box, because the item needs to know what list to save to. But it's not hard to fix that!

We start, as always, with a test. Just to illustrate what the problem is, let's see what happens if we just try to call `form.save()`:

src/lists/tests/test_forms.py (ch15l033)

```
    def test_form_save_handles_saving_to_a_list(self):
        form = ItemForm(data={"text": "do me"})
        new_item = form.save()
```

Django isn't happy, because an item needs to belong to a list:

```
django.db.utils.IntegrityError: NOT NULL constraint failed: lists_item.list_id
```

Our solution is to tell the form's save method what list it should save to:

src/lists/tests/test_forms.py (ch15l034)

```
from lists.models import Item, List
[...]

    def test_form_save_handles_saving_to_a_list(self):
        mylist = List.objects.create()
        form = ItemForm(data={"text": "do me"})
        new_item = form.save(for_list=mylist)  ❶
        self.assertEqual(new_item, Item.objects.get())  ❷
        self.assertEqual(new_item.text, "do me")
        self.assertEqual(new_item.list, mylist)
```

❶ We'll imagine that the `.save()` method takes a `for_list=` argument.

❷ We then make sure that the item is correctly saved to the database, with the right attributes.

The tests fail as expected, because as usual, it's still only wishful thinking:

```
    new_item = form.save(for_list=mylist)
TypeError: BaseModelForm.save() got an unexpected keyword argument 'for_list'
```

Here's how we can implement a custom save method:

src/lists/forms.py (ch15l035)

```
class ItemForm(forms.models.ModelForm):
    class Meta:
        [...]

    def save(self, for_list):
        self.instance.list = for_list
        return super().save()
```

The `.instance` attribute on a form represents the database object that is being modified or created. And I only learned that as I was writing this chapter! There are other ways of getting this to work, including manually creating the object yourself, or using the `commit=False` argument to save, but this way seemed neatest. We'll explore a different way of making a form "know" what list it's for in the next chapter. A quick test run to prove it works:

```
Ran 22 tests in 0.086s

OK
```

Finally, we can refactor our views. `new_list()` first:

src/lists/views.py (ch15l036)

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        nulist = List.objects.create()
        form.save(for_list=nulist)
        return redirect(nulist)
    else:
        return render(request, "home.html", {"form": form})
```

Rerun the test to check that everything still passes:

```
Ran 22 tests in 0.086s
OK
```

Then, refactor `view_list()`:

src/lists/views.py (ch15l037)

```
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    form = ItemForm()

    if request.method == "POST":
        form = ItemForm(data=request.POST)
        if form.is_valid():
            form.save(for_list=our_list)
            return redirect(our_list)

    return render(request, "list.html", {"list": our_list, "form": form})
```

We still have full passes:

```
Ran 22 tests in 0.111s
OK
```

And:

```
Ran 4 tests in 14.367s
OK
```

Great! Let's commit our changes:

```
$ git commit -am "implement custom save method for the form"
```

Our two views are now looking very much like "normal" Django views: they take information from a user's request, combine it with some custom logic or information from the URL (`list_id`), pass it to a form for validation and possible saving, and then redirect or render a template.

Forms and validation are really important in Django—and in web programming, in general—so let's try to make a slightly more complicated one in the next chapter, to learn how to prevent duplicate items.

Tips

Thin views
: If you find yourself looking at complex views, and having to write a lot of tests for them, it's time to start thinking about whether that logic could be moved elsewhere: possibly to a form, like we've done here. Another possible place would be a custom method on the model class—and, once the complexity of the app demands it, out of Django-specific files and into your own classes and functions, that capture your core business logic.

Each test should test one thing
: The heuristic is to be suspicious if there's more than one assertion in a unit test. Sometimes two assertions are closely related, so they belong together. But often your first draft of a test ends up testing multiple behaviours. Therefore, it's worth rewriting it as several tests so that each one can pinpoint specific problems more precisely, and so one failure doesn't mask another. Helper functions can keep your tests from getting too bloated.

Be aware of trade-offs when using frameworks
: When we switched to using a `ModelForm`, we saw that it forced us to change the `name=` attribute in our frontend HTML. Django gave us a lot: it autogenerated the form based on the model, and we have a nice API for doing both validation and saving objects. But we lost something too—we'll revisit this trade-off in the next chapter.

CHAPTER 16
More Advanced Forms

Let's look at some more advanced forms usage. We've successfully helped our users to avoid blank list items, so now let's help them to avoid duplicate items as well.

Our validation constraint so far has been about preventing blank items, and as you may remember, it turned out that we can enforce that very easily in the frontend. Avoiding duplicate items, however, is less straightforward to do in the frontend (although not impossible, of course), so this chapter will lean more heavily on server-side validation, and bubbling errors from the backend back up to the UI.

This chapter goes into the more intricate details of Django's forms framework, so you have my official permission to skim through it if you already know all about customising Django forms and how to display errors in the UI, or if you're reading this book for the TDD rather than for the Django.

If you're still learning Django, there's good stuff in here! If you want to just skim-read, that's OK too. Make sure you take a quick look at "An Aside on When to Test for Developer Silliness" on page 366, and "Recap: What to Test in Views" on page 395 at the end.

Another FT for Duplicate Items

We add a second test method to `ItemValidationTest`, and tell a little story about what we want to see happen when a user tries to enter the same item twice into their to-do list:

src/functional_tests/test_list_item_validation.py (ch16l001)

```
def test_cannot_add_duplicate_items(self):
    # Edith goes to the home page and starts a new list
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys("Buy wellies")
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table("1: Buy wellies")

    # She accidentally tries to enter a duplicate item
    self.get_item_input_box().send_keys("Buy wellies")
    self.get_item_input_box().send_keys(Keys.ENTER)

    # She sees a helpful error message
    self.wait_for(
        lambda: self.assertEqual(
            self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback").text,
            "You've already got this in your list",
        )
    )
```

Why use two test methods instead of extending one, or instead of creating a new file and class? It's a judgement call. These two feel closely related; they're both about validation on the same input field, so it feels right to keep them in the same file. On the other hand, they're logically separate enough that it's practical to keep them in different methods:

```
$ python src/manage.py test functional_tests.test_list_item_validation
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; [...]

Ran 2 tests in 9.613s
```

OK, so we know the first of the two tests passes now. Is there a way to run just the failing one, I hear you ask? Why, yes indeed:

```
$ python src/manage.py test functional_tests.\
test_list_item_validation.ItemValidationTest.test_cannot_add_duplicate_items
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; [...]
```

In any case, let's commit it:

```
$ git commit -am"Ft for duplicate item validation"
```

Preventing Duplicates at the Model Layer

So, if we want to start to implement our actual objective for the chapter, let's write a new test that checks that duplicate items in the same list raise an error:

src/lists/tests/test_models.py (ch16l002)

```
def test_duplicate_items_are_invalid(self):
    mylist = List.objects.create()
    Item.objects.create(list=mylist, text="bla")
    with self.assertRaises(ValidationError):
        item = Item(list=mylist, text="bla")
        item.full_clean()
```

And, while it occurs to us, we add another test to make sure we don't overdo it on our integrity constraints:

src/lists/tests/test_models.py (ch16l003)

```
def test_CAN_save_same_item_to_different_lists(self):
    list1 = List.objects.create()
    list2 = List.objects.create()
    Item.objects.create(list=list1, text="bla")
    item = Item(list=list2, text="bla")
    item.full_clean()  # should not raise
```

I always like to put a little comment for tests that are checking that a particular use case should *not* raise an error; otherwise, it can be hard to see what's being tested:

```
AssertionError: ValidationError not raised
```

If we want to get it deliberately wrong, we can do this:

src/lists/models.py (ch16l004)

```
class Item(models.Model):
    text = models.TextField(default="", unique=True)
    list = models.ForeignKey(List, default=None, on_delete=models.CASCADE)
```

That lets us check that our second test really does pick up on this problem:

```
ERROR: test_CAN_save_same_item_to_different_lists (lists.tests.test_models.List
AndItemModelsTest.test_CAN_save_same_item_to_different_lists)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/lists/tests/test_models.py", line 59, in
test_CAN_save_same_item_to_different_lists
    item.full_clean()  # should not raise
    [...]
django.core.exceptions.ValidationError: {'text': ['Item with this Text already
exists.']}
[...]
```

An Aside on When to Test for Developer Silliness

One of the judgement calls in testing is when you should write tests that sound like "check that we haven't done something weird". In general, you should be wary of these.

In this case, we've written a test to check that you can't save duplicate items to the same list. Now, the simplest way to get that test to pass, the way in which you'd write the fewest lines of code, would be to make it impossible to save *any* duplicate items. That justifies writing another test, despite the fact that it would be a "silly" or "wrong" thing for us to code.

But you can't be writing tests for every possible way we could have coded something wrong.[1] If you have a function that adds two numbers, you can write a couple of tests:

```
assert adder(1, 1) == 2
assert adder(2, 1) == 3
```

But you have the right to assume that the implementation isn't deliberately screwy or perverse:

```
def adder(a, b):
    # unlikely code!
    if a == 3:
        return 666
    else:
        return a + b
```

One way of putting it is: trust yourself not to do something *deliberately* silly, but do protect against things that might be *accidentally* silly.

1 With that said, you can come pretty close. Once you get comfortable writing tests manually, take a look at Hypothesis. It lets you automatically generate input for your tests, covering many more test scenarios than you could realistically type manually. It's not always easy to see how to use it, but for the right kind of problem, it can be very powerful; the very first time I used it, it found a bug!

Just like `ModelForm`, models can use an inner class called `Meta`, and that's where we can implement a constraint that says an item must be unique for a particular list—or, in other words, that `text` and `list` must be unique together:

src/lists/models.py (ch16l005)

```
class Item(models.Model):
    text = models.TextField(default="")
    list = models.ForeignKey(List, default=None, on_delete=models.CASCADE)

    class Meta:
        unique_together = ("list", "text")
```

And that passes:

```
Ran 24 tests in 0.024s

OK
```

You might want to take a quick peek at the Django docs on model `Meta` attributes at this point.

Rewriting the Old Model Test

That long-winded model test did serendipitously help us find unexpected bugs, but now it's time to rewrite it. I wrote it in a very verbose style to introduce the Django ORM, but in fact, we can get the same coverage from a couple of much shorter tests. Delete `test_saving_and_retrieving_items` and replace it with this:

src/lists/tests/test_models.py (ch16l006)

```
class ListAndItemModelsTest(TestCase):
    def test_default_text(self):
        item = Item()
        self.assertEqual(item.text, "")

    def test_item_is_related_to_list(self):
        mylist = List.objects.create()
        item = Item()
        item.list = mylist
        item.save()
        self.assertIn(item, mylist.item_set.all())

    [...]
```

That's more than enough really—a check of the default values of attributes on a freshly initialised model object is enough to sense-check that we've probably set some fields up in *models.py*. The "item is related to list" test is a real "belt and braces" test to make sure that our foreign key relationship works.

While we're at it, we can split this file out into tests for `Item` and tests for `List` (there's only one of the latter, `test_get_absolute_url`):

src/lists/tests/test_models.py (ch16l007)

```
class ItemModelTest(TestCase):
    def test_default_text(self):
        [...]


class ListModelTest(TestCase):
    def test_get_absolute_url(self):
        [...]
```

That's neater and tidier:

```
$ python src/manage.py test lists
[...]
Ran 25 tests in 0.092s

OK
```

Integrity Errors That Show Up on Save

A final aside before we move on. Do you remember the discussion mentioned in Chapter 14 that some data integrity errors *are* picked up on save? It all depends on whether the integrity constraint is actually being enforced by the database.

Try running `makemigrations` and you'll see that Django wants to add the `unique_together` constraint to the database itself, rather than just having it as an application-layer constraint:

```
$ python src/manage.py makemigrations
Migrations for 'lists':
  src/lists/migrations/0005_alter_item_unique_together.py
    ~ Alter unique_together for item (1 constraint(s))
```

Now let's run the migration:

```
$ python src/manage.py migrate
```

What to Do If You See an IntegrityError When Running Migrations

When you run the migration, you may encounter the following error:

```
$ python src/manage.py migrate
Operations to perform:
  Apply all migrations: auth, contenttypes, lists, sessions
Running migrations:
  Applying lists.0005_alter_item_unique_together...
Traceback (most recent call last):
[...]
sqlite3.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text

[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

The problem is that we have at least one database record that *used* to be valid but, after introducing our new constraint—the `unique_together`—it's no longer compatible.

To fix this problem locally, we can just delete `src/db.sqlite3` and run the migration again. We can do this because the database on our laptop is only used for dev, so the data in it is not important.

In Chapter 18, we'll deploy our new code to production, and discuss what to do if we run into migrations and data integrity issues at that point.

Now, if we change our duplicate test to do a `.save` instead of a `.full_clean`…

src/lists/tests/test_models.py (ch16l008)

```
def test_duplicate_items_are_invalid(self):
    mylist = List.objects.create()
    Item.objects.create(list=mylist, text="bla")
    with self.assertRaises(ValidationError):
        item = Item(list=mylist, text="bla")
        # item.full_clean()
        item.save()
```

It gives:

```
ERROR: test_duplicate_items_are_invalid
(lists.tests.test_models.ItemModelTest.test_duplicate_items_are_invalid)
[...]
sqlite3.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

You can see that the error bubbles up from SQLite, and it's a different error from the one we want—an `IntegrityError` instead of a `ValidationError`.

Let's revert our changes to the test, and see them all passing again:

```
$ python src/manage.py test lists
[...]
Ran 25 tests in 0.092s
OK
```

And now it's time to commit our model-layer changes:

```
$ git status # should show changes to tests + models and new migration
$ git add src/lists
$ git diff --staged
$ git commit -m "Implement duplicate item validation at model layer"
```

Experimenting with Duplicate Item Validation at the Views Layer

Let's try running our FT, to see if that's made any difference.

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: .invalid-feedback; [...]
```

In case you didn't see it as it flew past, the site is 500ing,[2] as in Figure 16-1 (feel free to try it out manually).

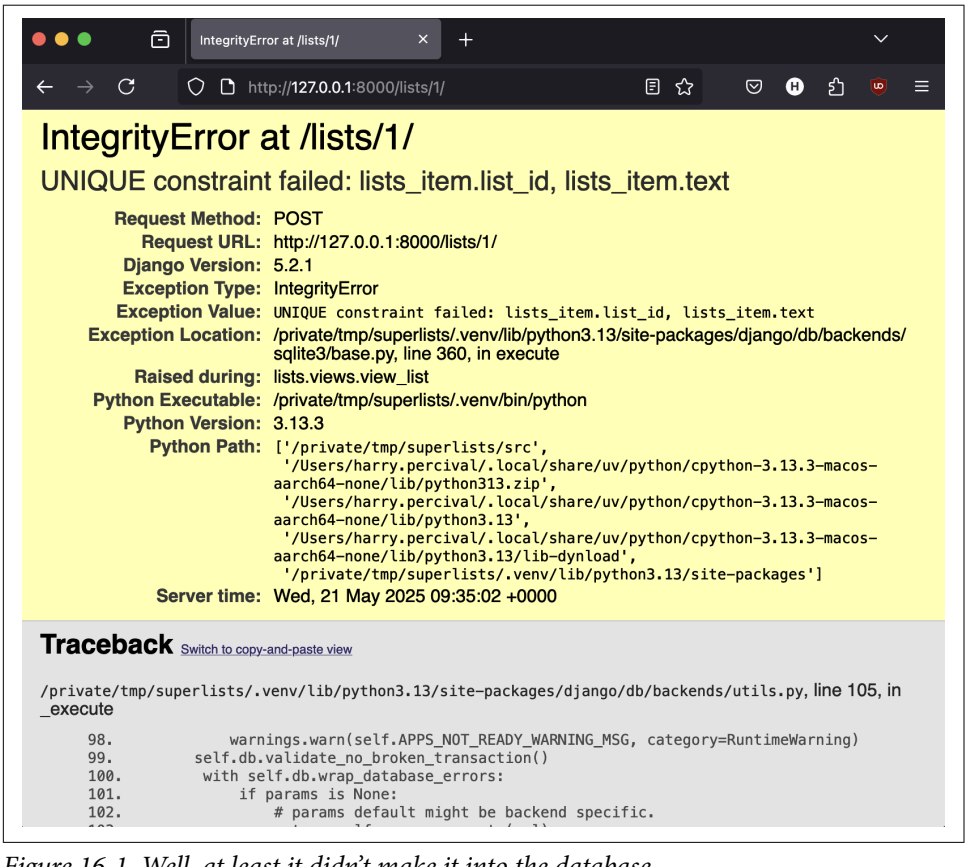


Figure 16-1. Well, at least it didn't make it into the database

2 500ing, showing a server error, code 500—of course you can use HTTP status codes as verbs!

We need to be clearer on what we want to happen at the views level. Let's write a unit test to set out our expectations:

src/lists/tests/test_views.py (ch16l009)

```
class ListViewTest(TestCase):
    [...]
    def test_for_invalid_input_nothing_saved_to_db(self):
        [...]
    def test_for_invalid_input_renders_list_template(self):
        [...]
    def test_for_invalid_input_shows_error_on_page(self):
        [...]

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        list1 = List.objects.create()
        Item.objects.create(list=list1, text="textey")

        response = self.client.post(
            f"/lists/{list1.id}/",
            data={"text": "textey"},
        )

        expected_error = html.escape("You've already got this in your list")
        self.assertContains(response, expected_error)  ❶
        self.assertTemplateUsed(response, "list.html")  ❷
        self.assertEqual(Item.objects.all().count(), 1)  ❸
```

❶ Here's our main assertion, which is that we want to see a nice error message on the page.

❷ Here's where we check that it's landing on the normal list page.

❸ And we double-check that we haven't saved anything to the database.[3]

That test confirms that the `IntegrityError` is bubbling all the way up:

```
  File "...goat-book/src/lists/views.py", line 28, in view_list
    form.save(for_list=our_list)
    ~~~~~~~~~^^^^^^^^^^^^^^^^^^^
[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

3 Harry, didn't we spend time in the last chapter making sure all the asserts were in different tests? Absolutely yes. Feel free to do that! If I had to justify myself, I'd say that we already have all the granular asserts for *one* error type, and this really is just a smoke test that an additional error type is also handled. So, arguably, it doesn't need to be so granular.

We want to avoid integrity errors! Ideally, we want the call to `is_valid()` to somehow notice the duplication error before we even try to save. But to do that, our form will need to know in advance what list it's being used for. Let's put a skip on this test for now:

src/lists/tests/test_views.py (ch16l010)

```
from unittest import skip
[...]

    @skip
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
```

A More Complex Form to Handle Uniqueness Validation

The form to create a new list only needs to know one thing: the new item text. A form validating that list items are unique will need to know what list they're in as well. Just as we overrode the save method on our `ItemForm`, this time we'll override the *constructor* on our new form class so that it knows what list it applies to.

Let's duplicate our tests from the previous form, tweaking them slightly:

src/lists/tests/test_forms.py (ch16l011)

```
[...]
from lists.forms import (
    DUPLICATE_ITEM_ERROR,
    EMPTY_ITEM_ERROR,
    ExistingListItemForm,
    ItemForm,
)
[...]

class ExistingListItemFormTest(TestCase):
    def test_form_renders_item_text_input(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_)  ❶
        self.assertIn('placeholder="Enter a to-do item"', form.as_p())

    def test_form_validation_for_blank_items(self):
        list_ = List.objects.create()
        form = ExistingListItemForm(for_list=list_, data={"text": ""})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors["text"], [EMPTY_ITEM_ERROR])

    def test_form_validation_for_duplicate_items(self):
        list_ = List.objects.create()
        Item.objects.create(list=list_, text="no twins!")
        form = ExistingListItemForm(for_list=list_, data={"text": "no twins!"})
        self.assertFalse(form.is_valid())
        self.assertEqual(form.errors["text"], [DUPLICATE_ITEM_ERROR])
```

❶ We're specifying that our new `ExistingListItemForm` will take an argument `for_list=` in its constructor, to be able to specify which list the item is for.

Next we iterate through a few TDD cycles until we get a form with a custom constructor, which just ignores its `for_list` argument. (I won't show them all, but I'm sure you'll do them, right? Remember, the Goat sees all.)

src/lists/forms.py (ch16l012)

```
DUPLICATE_ITEM_ERROR = "You've already got this in your list"
[...]
class ExistingListItemForm(forms.models.ModelForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

At this point, our error should be:

```
ValueError: ModelForm has no model class specified.
```

Then, let's see if making it inherit from our existing form helps:

src/lists/forms.py (ch16l013)

```
class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

Yes, that takes us down to just one failure:

```
FAIL: test_form_validation_for_duplicate_items (lists.tests.test_forms.Existing
ListItemFormTest.test_form_validation_for_duplicate_items)
[...]
    self.assertFalse(form.is_valid())
AssertionError: True is not false
```

The next step requires a little knowledge of Django's validation system—you can read up on it in the Django docs on model validation and form validation.

We can customise validation for a field by implementing a `clean_<fieldname>()` method, and raising a `ValidationError` if the field is invalid:

src/lists/forms.py (ch16l013-1)

```
from django.core.exceptions import ValidationError
[...]

class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.instance.list = for_list

    def clean_text(self):
        text = self.cleaned_data["text"]
        if self.instance.list.item_set.filter(text=text).exists():
            raise forms.ValidationError(DUPLICATE_ITEM_ERROR)
        return text
```

That makes the tests happy:

```
Found 29 test(s).
[...]
OK (skipped=1)
```

We're there! A quick commit:

```
$ git diff
$ git add src/lists/forms.py src/lists/tests/test_forms.py
$ git commit -m "implement ExistingListItemForm, add DUPLICATE_ITEM_ERROR message"
```

Using the Existing List Item Form in the List View

Now let's see if we can put this form to work in our view. We remove the skip and, while we're at it, we can use our new constant:

src/lists/tests/test_views.py (ch16l014)

```
from lists.forms import (
    DUPLICATE_ITEM_ERROR,
    EMPTY_ITEM_ERROR,
)
[...]

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        [...]
        expected_error = html.escape(DUPLICATE_ITEM_ERROR)
        self.assertContains(response, expected_error)
        [...]
```

We see our `IntegrityError` once again:

```
django.db.utils.IntegrityError: UNIQUE constraint failed: lists_item.list_id,
lists_item.text
```

Our fix for this is to switch to using the new form class:

src/lists/views.py (ch16l016)

```
from lists.forms import ExistingListItemForm, ItemForm
[...]
def view_list(request, list_id):
    our_list = List.objects.get(id=list_id)
    form = ExistingListItemForm(for_list=our_list)  ❶

    if request.method == "POST":
        form = ExistingListItemForm(for_list=our_list, data=request.POST)  ❶
        if form.is_valid():
            form.save(for_list=our_list)  ❷
            [...]
    [...]
```

❶ We swap out `ItemForm` for `ExistingListItemForm`, and pass in the `for_list=`.

❷ This is a bit annoying—we're duplicating the `for_list=` argument. This form should already know this!

Customising the Save Method on Our New Form

Programming by wishful thinking, as always. Let's specify in our *views.py* that we wish we could call `save()` without the duplicated argument:

src/lists/views.py (ch16l016-1)

```
@@ -25,6 +25,6 @@ def view_list(request, list_id):
     if request.method == "POST":
         form = ExistingListItemForm(for_list=our_list, data=request.POST)
         if form.is_valid():
-            form.save(for_list=our_list)
+            form.save()
             return redirect(our_list)
     return render(request, "list.html", {"list": our_list, "form": form})
```

That gives us a failure as expected:

```
  File "...goat-book/src/lists/views.py", line 28, in view_list
    form.save()
    ~~~~~~~~~^^
TypeError: ItemForm.save() missing 1 required positional argument: 'for_list'
```

Let's drop down to the forms level, and write another unit test for how we want our save method to work:

src/lists/tests/test_forms.py (ch16l017)

```
class ExistingListItemFormTest(TestCase):
[...]
    def test_form_save(self):
        mylist = List.objects.create()
        form = ExistingListItemForm(for_list=mylist, data={"text": "hi"})
        self.assertTrue(form.is_valid())
        new_item = form.save()
        self.assertEqual(new_item, Item.objects.get())
[...]
```

We can make our form call the grandparent save method:

src/lists/forms.py (ch16l018)

```
class ExistingListItemForm(ItemForm):
    [...]
    def save(self):
        return forms.models.ModelForm.save(self)  ❶
```

❶ This manually calls the grandparent `save()`. Personal opinion here: I could have used `super()`, but I prefer not to use `super()` when it requires arguments, say, to get a grandparent. I find Python 3's `super()` with no arguments is awesome to get the immediate parent. Anything else is too error-prone—and, besides, I find it ugly. YMMV.

OK, how does that look? Yep, both the forms level and views level tests now pass:

```
$ python src/manage.py test lists
[...]
Ran 30 tests in 0.082s

OK
```

Time to see what our FTs think!

The FTs Pick Up an Issue with Bootstrap Classes

Unfortunately, the FTs are telling us we're not done:

```
$ python src/manage.py test functional_tests.test_list_item_validation
[...]
FAIL: test_cannot_add_duplicate_items [...]
----------------------------------------------------------------------
[...]
AssertionError: '' != "You've already got this in your list"
+ You've already got this in your list
```

Let's spin up the server with `runserver` and try it out manually—with DevTools open—to see what's going on. If you look through the HTML, you'll see our error `div` is there, with the correct error text, but it's greyed out, indicating that it's hidden (as in Figure 16-2).

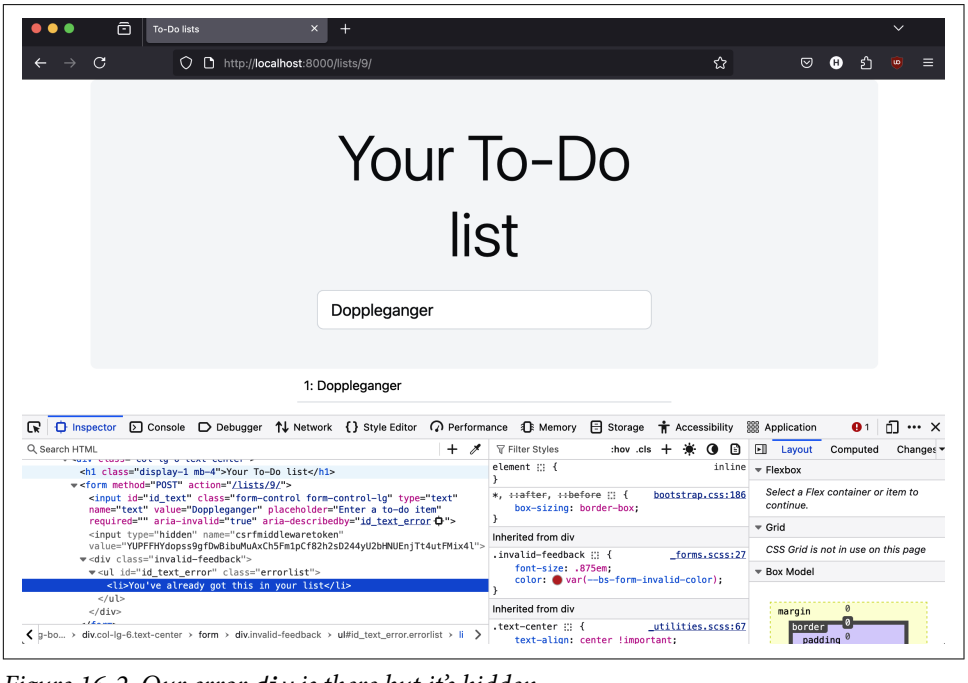


Figure 16-2. Our error `div` is there but it's hidden

I had to dig through the docs a little, but it turns out that Bootstrap requires form elements with errors to have *another* custom class, `is-invalid`. You can actually try this out in DevTools! If you double-click, you can edit the HTML and add the class, as in Figure 16-3.

Figure 16-3. Hack it in manually—yay

Conditionally Customising CSS Classes for Invalid Forms

Speaking of hackery, I'm starting to get a bit nervous about the amount of hackery we're doing in our forms now, but let's try getting this to work by doing *even more* customisation in our forms.

We want this behaviour for both types of form really, so it can go in the tests for the parent `ItemForm` class:

src/lists/tests/test_forms.py (ch16l019-1)

```
class ItemFormTest(TestCase):
    def test_form_item_input_has_placeholder_and_css_classes(self):
        [...]
    def test_form_validation_for_blank_items(self):
        [...]

    def test_invalid_form_has_bootstrap_is_invalid_css_class(self):
        form = ItemForm(data={"text": ""})
        self.assertFalse(form.is_valid())
        field = form.fields["text"]
        self.assertEqual(
            field.widget.attrs["class"],  ❶
            "form-control form-control-lg is-invalid",
        )

    def test_form_save_handles_saving_to_a_list(self):
        [...]
```

❶ Here's where you can inspect the `class` attribute on the input field `widget`.

And here's how we can make it work, by overriding the `is_valid()` method:

src/lists/forms.py (ch16l019-2)

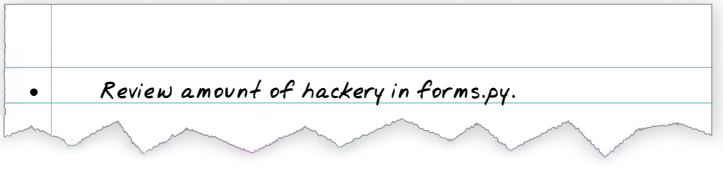
```
class ItemForm(forms.models.ModelForm):
    class Meta:
        [...]

    def is_valid(self):
        result = super().is_valid()  ❶
        if not result:
            self.fields["text"].widget.attrs["class"] += " is-invalid"  ❷
        return result  ❸

    def save(self, for_list):
        [...]
```

❶ We make sure to call the parent `is_valid()` method first, so we can do all the normal built-in validation.

❷ Here's how we add the extra CSS class to our `widget`.

❸ And we remember to return the result.

It's not *too* bad—but, as I say, I'm getting nervous about the amount of fiddly code in our forms classes. Let's make a note on our scratchpad, and come back to it when our FT is passing perhaps:



Speaking of our FT, let's see how it does now:

```
$ python src/manage.py test functional_tests.test_list_item_validation
[...]
======================================================================
FAIL: test_cannot_add_empty_list_items (functional_tests.test_list_item_validat
ion.ItemValidationTest.test_cannot_add_empty_list_items)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/functional_tests/test_list_item_validation.py", line
47, in test_cannot_add_empty_list_items
    self.wait_for_row_in_list_table("2: Make tea")
  File "...goat-book/src/functional_tests/base.py", line 37, in
wait_for_row_in_list_table
    self.assertIn(row_text, [row.text for row in rows])
AssertionError: '2: Make tea' not found in ['1: Make tea', '2: Purchase milk']
```

Ooops; what happened here (Figure 16-4)?

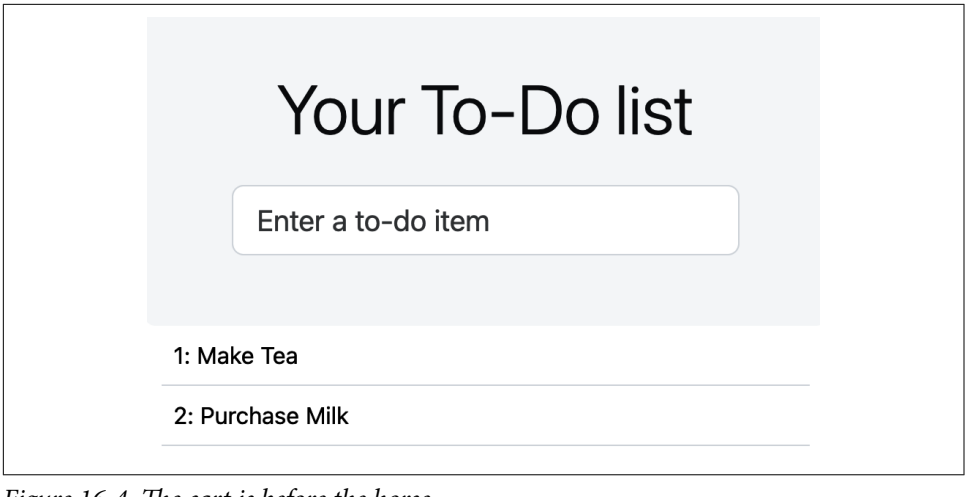


Figure 16-4. The cart is before the horse

A Little Digression on Queryset Ordering and String Representations

Something seems to be going wrong with the ordering of our list items. Trying to fix this by iterating against an FT is going to be slow, so let's work at the unit test level.

We'll add a test that checks that list items are ordered in the sequence they are inserted. You'll have to forgive me if I jump straight to the right answer, using intuition borne of long experience, but I suspect that it might be sorting alphabetically based on list text instead (what else would it sort by after all?), so I'll pick some text values designed to test that hypothesis:

src/lists/tests/test_models.py (ch16l020)

```
class ListModelTest(TestCase):
    def test_get_absolute_url(self):
        [...]

    def test_list_items_order(self):
        list1 = List.objects.create()
        item1 = Item.objects.create(list=list1, text="i1")
        item2 = Item.objects.create(list=list1, text="item 2")
        item3 = Item.objects.create(list=list1, text="3")
        self.assertEqual(
            list1.item_set.all(),
            [item1, item2, item3],
        )
```

FTs are a slow feedback loop. Switch to unit tests when you want to drill down on edge case bugs.

That gives us a new failure, but it's not very readable:

```
AssertionError: <QuerySet [<Item: Item object (3)>, <Item[40 chars]2)>]> !=
[<Item: Item object (1)>, <Item: Item obj[29 chars](3)>]
```

We need a better string representation for our `Item` model. Let's add another unit test:

src/lists/tests/test_models.py (ch16l021)

```
class ItemModelTest(TestCase):
    [...]
    def test_string_representation(self):
        item = Item(text="some text")
        self.assertEqual(str(item), "some text")
```

Ordinarily, you would be wary of adding more failing tests when you already have some—it makes reading test output that much more complicated, and just generally makes you nervous. Will we ever get back to a working state? In this case, they're all quite simple tests, so I'm not worried.

That gives us:

```
AssertionError: 'Item object (None)' != 'some text'
```

And it also gives us the other two failures. Let's start fixing them all now:

src/lists/models.py (ch16l022)

```
class Item(models.Model):
    [...]

    def __str__(self):
        return self.text
```

Now we're down to one failure, and the ordering test has a more readable failure message:

```
AssertionError: <QuerySet [<Item: 3>, <Item: i1>, <Item: item 2>]> != [<Item:
i1>, <Item: item 2>, <Item: 3>]
```

That confirms our suspicion that the ordering was alphabetical.

We can fix that in the `class Meta`:

src/lists/models.py (ch16l023)

```
class Item(models.Model):
    [...]
    class Meta:
        ordering = ("id",)
        unique_together = ("list", "text")
```

Does that work?

```
AssertionError: <QuerySet [<Item: i1>, <Item: item 2>, <Item: 3>]> != [<Item:
i1>, <Item: item 2>, <Item: 3>]
```

Urp? It has worked; you can see the items *are* in the same order, but the tests are confused.

I keep running into this problem actually—Django QuerySets don't compare well with lists. We can fix it by converting the QuerySet to a list in our test:[4]

src/lists/tests/test_models.py (ch16l024)

```
        self.assertEqual(
            list(list1.item_set.all()),
            [item1, item2, item3],
        )
```

4 You could also check out `assertSequenceEqual` from `unittest`, and `assertQuerysetEqual` from Django's test tools—although I confess, when I last looked at `assertQuerysetEqual`, I was quite baffled…

That works; we get a fully passing unit test suite:

```
Ran 33 tests in 0.034s

OK
```

We do need a migration for that ordering change though:

```
$ python src/manage.py makemigrations
Migrations for 'lists':
  src/lists/migrations/0006_alter_item_options.py
    ~ Change Meta options on item
```

And as a final check, we rerun *all* the FTs:

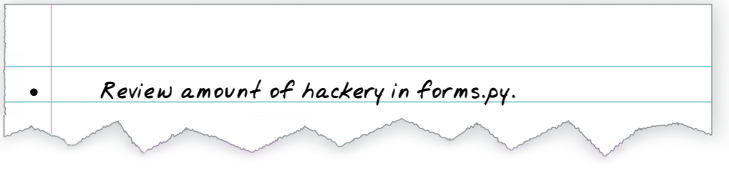
```
$ python src/manage.py test functional_tests
[...]
 ----------------------------------------------------------------------
Ran 5 tests in 19.048s

OK
```

Hooray! Time for a final commit:

```
git status
git add src
git commit -m "Add is-invalid css class, fix list item ordering"
```

On the Trade-offs of Django ModelForms, and Frameworks in General

Let's come back to our scratchpad item:



Let's take a look at the current state of our forms classes. We've got a real mix of presentation logic, validation logic, and ORM/storage logic:

src/lists/forms.py

```
class ItemForm(forms.models.ModelForm):
    class Meta:
        model = Item
        fields = ("text",)
        widgets = {
            "text": forms.widgets.TextInput(
                attrs={
                    "placeholder": "Enter a to-do item",  ❶
                    "class": "form-control form-control-lg",  ❶
                }
            ),
        }
        error_messages = {"text": {"required": EMPTY_ITEM_ERROR}}

    def is_valid(self):
        result = super().is_valid()
        if not result:
            self.fields["text"].widget.attrs["class"] += " is-invalid"  ❶
        return result

    def save(self, for_list):  ❸
        self.instance.list = for_list
        return super().save()


class ExistingListItemForm(ItemForm):
    def __init__(self, for_list, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.instance.list = for_list  ❸

    def clean_text(self):
        text = self.cleaned_data["text"]
        if self.instance.list.item_set.filter(text=text).exists():  ❷
            raise forms.ValidationError(DUPLICATE_ITEM_ERROR)  ❷
        return text

    def save(self):
        return forms.models.ModelForm.save(self)  ❸
```

❶ Presentation logic

❷ Validation logic

❸ ORM/storage logic

I think what's happened is that we've reached the limits of the Django forms framework's sweet spot. `ModelForms` can be great *because* they can do presentation, validation, and database storage all in one go, so you can get a lot done without much code. But once you want to customise the default behaviours for each of those things, the code you *do* end up writing starts to get hard to understand.

Let's see what things would look like if we tried to:

1. Move the responsibility for presentation and the rendering of HTML back into the template.
2. Stop using `ModelForm` and do any database logic more explicitly, with less magic.

Another Flip-flop!

We spent most of the last chapter switching from handcrafted HTML to having our form autogenerated by Django, and now we're switching back. It's a little frustrating, and I could have gone back and changed the book's outline to avoid the back and forth, but I prefer to show software development as it really is.

We often try things out and end up changing our minds. Particularly with frameworks like Django, you can find yourself taking advantage of autogenerated shortcuts for as long as they work. But at some point, you meet the limits of what the framework designers have anticipated, and it's time to go back to doing the work yourself.

Frameworks have trade-offs. It doesn't mean you should always reinvent the wheel! It's OK to cut yourself some slack for "wasting time" on avenues that don't work out, or revisiting decisions that worked well in the past, but don't work so well now.

Moving Presentation Logic Back into the Template

We're talking about another refactor here; we want to move some functionality out of the form and into the template/views layer. How do we make sure we've got good test coverage?

- We currently have some tests for the CSS classes including `is-invalid` in *test_forms.py*.
- We have some tests of some form attributes in *test_views.py*—e.g., the asserts on the input's `name`.
- And the FTs, ultimately, will tell us if things "really work" or not, including testing the interaction between our HTML, Bootstrap, and the browser (e.g., CSS visibility).

What we are learning is that the things we're testing in *test_forms.py* will need to move.

Lower-level tests are good for exploring an API, but they are tightly coupled to it. Higher-level tests can enable more refactoring.

Here's one way to write that kind of test:

src/lists/tests/test_views.py (ch16l025-1)

```
class ListViewTest(TestCase):
    [...]
    def test_for_invalid_input_shows_error_on_page(self):
        [...]

    def test_for_invalid_input_sets_is_invalid_class(self):
        response = self.post_invalid_input()
        parsed = lxml.html.fromstring(response.content)
        [input] = parsed.cssselect("input[name=text]")
        self.assertIn("is-invalid", input.get("class"))

    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        [...]
```

That's green straight away:

```
Ran 34 tests in 0.040s

OK
```

As always, it's nice to deliberately break it, to see whether it has a nice failure message, if nothing else. Let's do that in *forms.py*:

src/lists/forms.py (ch16l025-2)

```
@@ -24,7 +24,7 @@ class ItemForm(forms.models.ModelForm):
     def is_valid(self):
         result = super().is_valid()
         if not result:
-            self.fields["text"].widget.attrs["class"] += " is-invalid"
+            self.fields["text"].widget.attrs["class"] += " boo!"
         return result

     def save(self, for_list):
```

Reassuringly, both our old test and the new one fail:

```
[...]
======================================================================
FAIL: test_invalid_form_has_bootstrap_is_invalid_css_class (lists.tests.test_fo
rms.ItemFormTest.test_invalid_form_has_bootstrap_is_invalid_css_class)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/lists/tests/test_forms.py", line 30, in
test_invalid_form_has_bootstrap_is_invalid_css_class
    self.assertEqual(
    ~~~~~~~~~~~~~~~~^
        field.widget.attrs["class"],
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^
        "form-control form-control-lg is-invalid",
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    )
    ^
AssertionError: 'form-control form-control-lg boo!' != 'form-control
form-control-lg is-invalid'
- form-control form-control-lg boo!
?                               ^^^^
+ form-control form-control-lg is-invalid
?                               ^^^^^^^^^^


======================================================================
FAIL: test_for_invalid_input_sets_is_invalid_class (lists.tests.test_views.List
ViewTest.test_for_invalid_input_sets_is_invalid_class)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/lists/tests/test_views.py", line 129, in
test_for_invalid_input_sets_is_invalid_class
    self.assertIn("is-invalid", input.get("class"))
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'is-invalid' not found in 'form-control form-control-lg boo!'

 ----------------------------------------------------------------------
Ran 34 tests in 0.039s

FAILED (failures=2)
```

Let's revert that and get back to passing.

So, rather than using the `{{ form.text }}` magic in our template, let's bring back our handcrafted HTML. It'll be longer, but at least all of our Bootstrap classes will be in one place, where we expect them, in the template:

src/lists/templates/base.html (ch16l025-4)

```
@@ -16,10 +16,22 @@
           <h1 class="display-1 mb-4">{% block header_text %}{% endblock %}</h1>

           <form method="POST" action="{% block form_action %}{% endblock %}" >
-            {{ form.text }}
             {% csrf_token %}
+            <input  ❶
+              id="id_text"
+              name="text"
+              class="form-control  ❷
+                     form-control-lg
+                     {% if form.errors %}is-invalid{% endif %}"
+              placeholder="Enter a to-do item"
+              value="{{ form.text.value | default:'' }}"  ❸
+              aria-describedby="id_text_feedback"  ❹
+              required
+            />
             {% if form.errors %}
-              <div class="invalid-feedback">{{ form.errors.text }}</div>
+              <div id="id_text_feedback" class="invalid-feedback">  ❹
+                {{ form.errors.text.0 }}  ❺
+              </div>
             {% endif %}
           </form>
         </div>
```

❶ Here's our artisan `<input>` once again, and the most important custom setting will be its `class` attributes.

❷ As you can see, we can use conditionals even for providing additional `class`-es.[5]

❸ The `| default` "filter" is a way to avoid the string "None" from showing up as the value in our input field.

❹ We add an `id` to the error message to be able to use `aria-describedby` on the input, as recommended in the Bootstrap docs; it makes the error message more accessible to screen readers.

❺ If you just try to use `form.errors.text`, you'll see that Django injects a `<ul>` list, because the forms framework can report multiple errors for each field. We know we've only got one, so we can use use `form.errors.text.0`.

5 We've split the `input` tag across multiple lines so it fits nicely on the screen. If you've not seen that before, it may look a little weird, but I promise it is valid HTML. You don't have to use it if you don't like it though.

That passes:

```
Ran 34 tests in 0.034s

OK
```

Out of curiosity, let's try a deliberate failure here:

src/lists/templates/base.html (ch16l025-5)

```
@@ -22,7 +22,7 @@
                name="text"
                class="form-control
                       form-control-lg
-                      {% if form.errors %}is-invalid{% endif %}"
+                      {% if form.errors %}isnt-invalid{% endif %}"
                placeholder="Enter a to-do item"
                value="{{ form.text.value | default:'' }}"
                aria-describedby="id_text_feedback"
```

The failure looks like this:

```
    self.assertIn("is-invalid", input.get("class"))
    ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'is-invalid' not found in 'form-control\n
form-control-lg\n                    isnt-invalid'
```

Hmm, that's not ideal actually. Let's tweak our assert:

src/lists/tests/test_views.py (ch16l025-6)

```
    def test_for_invalid_input_sets_is_invalid_class(self):
        response = self.post_invalid_input()
        parsed = lxml.html.fromstring(response.content)
        [input] = parsed.cssselect("input[name=text]")
        self.assertIn("is-invalid", set(input.classes))  ❶
```

❶ Rather than using `get("class")`, which returns a raw string, `lxml` can give us the classes as a list (well, actually a special object, but one that we can turn into a set).

That's more semantically correct, and gives a better error message:

```
        self.assertIn("is-invalid", set(input.classes))
        ~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: 'is-invalid' not found in {'form-control', 'isnt-invalid',
'form-control-lg'}
```

OK, that's good; we can revert the deliberate mistake in *base.html*.

Let's do a quick FT run to check we've got it right:

```
$ python src/manage.py test functional_tests.test_list_item_validation
Found 2 test(s).
[...]
OK
```

Good!

Tidying Up the Forms

Now let's start tidying up our forms. We can start by deleting the three presentation-layer tests from `ItemFormTest`:

src/lists/tests/test_forms.py (ch16l026)

```
@@ -10,28 +10,11 @@ from lists.models import Item, List


 class ItemFormTest(TestCase):
-    def test_form_item_input_has_placeholder_and_css_classes(self):
-        form = ItemForm()
-
-        rendered = form.as_p()
-
-        self.assertIn('placeholder="Enter a to-do item"', rendered)
-        self.assertIn('class="form-control form-control-lg"', rendered)
-
     def test_form_validation_for_blank_items(self):
         form = ItemForm(data={"text": ""})
         self.assertFalse(form.is_valid())
         self.assertEqual(form.errors["text"], [EMPTY_ITEM_ERROR])

-    def test_invalid_form_has_bootstrap_is_invalid_css_class(self):
-        form = ItemForm(data={"text": ""})
-        self.assertFalse(form.is_valid())
-        field = form.fields["text"]
-        self.assertEqual(
-            field.widget.attrs["class"],
-            "form-control form-control-lg is-invalid",
-        )
-
     def test_form_save_handles_saving_to_a_list(self):
         mylist = List.objects.create()
         form = ItemForm(data={"text": "do me"})
@@ -42,11 +25,6 @@ class ItemFormTest(TestCase):


 class ExistingListItemFormTest(TestCase):
-    def test_form_renders_item_text_input(self):
-        list_ = List.objects.create()
-        form = ExistingListItemForm(for_list=list_)
-        self.assertIn('placeholder="Enter a to-do item"', form.as_p())
-
     def test_form_validation_for_blank_items(self):
         list_ = List.objects.create()
         form = ExistingListItemForm(for_list=list_, data={"text": ""})
```

And now we can remove all that custom logic from the base `ItemForm` class:

src/lists/forms.py (ch16l027)

```
@@ -11,22 +11,8 @@ class ItemForm(forms.models.ModelForm):
     class Meta:
         model = Item
         fields = ("text",)
-        widgets = {
-            "text": forms.widgets.TextInput(
-                attrs={
-                    "placeholder": "Enter a to-do item",
-                    "class": "form-control form-control-lg",
-                }
-            ),
-        }
         error_messages = {"text": {"required": EMPTY_ITEM_ERROR}}

-    def is_valid(self):
-        result = super().is_valid()
-        if not result:
-            self.fields["text"].widget.attrs["class"] += " is-invalid"
-        return result
-
     def save(self, for_list):
         self.instance.list = for_list
         return super().save()
```

Deleting code, yay!

At this point we should be down to 31 passing tests:

```
Ran 31 tests in 0.024s

OK
```

Switching Back to Simple Forms

Now let's change our forms away from being `ModelForms` and back to regular forms. We'll keep the `save()` methods for now, but we'll switch to using the ORM more explicitly, rather than relying on the `ModelForm` magic:

src/lists/forms.py (ch16l028)

```
@@ -7,27 +7,29 @@ EMPTY_ITEM_ERROR = "You can't have an empty list item"
 DUPLICATE_ITEM_ERROR = "You've already got this in your list"


-class ItemForm(forms.models.ModelForm):
-    class Meta:
-        model = Item
-        fields = ("text",)
-        error_messages = {"text": {"required": EMPTY_ITEM_ERROR}}
+class ItemForm(forms.Form):
+    text = forms.CharField(
+        error_messages={"required": EMPTY_ITEM_ERROR},
+        required=True,
+    )

     def save(self, for_list):
-        self.instance.list = for_list
-        return super().save()
+        return Item.objects.create(
+            list=for_list,
+            text=self.cleaned_data["text"],
+        )


 class ExistingListItemForm(ItemForm):
     def __init__(self, for_list, *args, **kwargs):
         super().__init__(*args, **kwargs)
-        self.instance.list = for_list
+        self._for_list = for_list

     def clean_text(self):
         text = self.cleaned_data["text"]
-        if self.instance.list.item_set.filter(text=text).exists():
+        if self._for_list.item_set.filter(text=text).exists():
             raise forms.ValidationError(DUPLICATE_ITEM_ERROR)
         return text

     def save(self):
-        return forms.models.ModelForm.save(self)
+        return super().save(for_list=self._for_list)
```

We should still have passing tests at this point:

```
Ran 31 tests in 0.026s

OK
```

And we're in a better place I think!

Wrapping Up: What We've Learned About Testing Django

We're now at a point where our app looks a lot more like a "standard" Django app, and it implements the three common Django layers: models, forms, and views. We no longer have any "training wheel" tests, and our code looks pretty much like code we'd be happy to see in a real app.

We have one unit test file for each of our key source code files. Here's a recap of the biggest (and highest-level) one: *test_views*.

Recap: What to Test in Views

By way of a recap, let's see an outline of all the test methods and main assertions in our `test_views`. This isn't to say you should copy-paste these exactly—it's more like a list of things you should at least consider testing:

src/lists/tests/test_views.py, selected test methods and asserts

```
class ListViewTest(TestCase):
    def test_uses_list_template(self):
        response = self.client.get(f"/lists/{mylist.id}/")  ❶
        self.assertTemplateUsed(response, "list.html")  ❷
    def test_renders_input_form(self):
        parsed = lxml.html.fromstring(response.content)  ❸
        self.assertIn("text", [input.get("name") for input in inputs])  ❸
    def test_displays_only_items_for_that_list(self):
        self.assertContains(response, "itemey 1")  ❹
        self.assertContains(response, "itemey 2")  ❹
        self.assertNotContains(response, "other list item")  ❹
    def test_can_save_a_POST_request_to_an_existing_list(self):
        self.assertEqual(new_item.text, "A new item for an existing list")  ❺
    def test_POST_redirects_to_list_view(self):
        self.assertRedirects(response, f"/lists/{correct_list.id}/")  ❺
    def test_for_invalid_input_nothing_saved_to_db(self):
        self.assertEqual(Item.objects.count(), 0)  ❻
    def test_for_invalid_input_renders_list_template(self):
        self.assertEqual(response.status_code, 200)  ❻
        self.assertTemplateUsed(response, "list.html")  ❻
    def test_for_invalid_input_shows_error_on_page(self):
        self.assertContains(response, html.escape(EMPTY_ITEM_ERROR))  ❻
    def test_duplicate_item_validation_errors_end_up_on_lists_page(self):
        self.assertContains(response, expected_error)  ❼
        self.assertTemplateUsed(response, "list.html")  ❼
        self.assertEqual(Item.objects.all().count(), 1)  ❼
```

❶ Use the Django test client.

❷ Optionally (this is a bit of an implementation detail), check the template used.

❸ Check that key parts of your HTML are present. Things that are critical to the integration of frontend and backend are good candidates, like form action and input `name` attributes. Using `lxml` might be overkill, but it does give you less brittle tests.

❹ Think about smoke-testing any other template contents, or any logic in the template: any `{% for %}` or `{% if %}` might deserve a check.

❺ For POST requests, test the valid case via its database side effects, and the redirect response.

❻ For invalid requests, it's worth a basic check that errors make it back to the template.

❼ You don't *always* have to have ultra-granular tests though.

Next, we'll try to make our data validation more friendly by using a bit of client-side code. Uh-oh, you know what that means…

PART IV

More Advanced Topics in Testing

"Oh my gosh, what? Another section? Harry, I'm exhausted. It's already been four hundred pages; I don't think I can handle a whole nother section of the book. Particularly not if it's called 'Advanced'…maybe I can get away with just skipping it?"

Oh no, you can't! This may be called the "advanced" section, but it's full of really important topics for test-driven development (TDD) and web development. No way can you skip it. If anything, it's *even more important* than the first two sections.

First off, we'll get into that sine qua non of web development: JavaScript. Seeing how TDD works in another language can give you a whole new perspective.

We'll be talking about a key technique, "spiking", which is where you relax the strict rules of TDD and allow yourself a bit of exploratory hacking.

A common objection to TDD is "how can I write tests if I don't even know what I'm doing?" Spiking is the bit where you get to play around and figure things out, so you can come back and do it test-first later.

We'll be talking about how to integrate third-party systems, and how to test them. We'll cover mocking, which is hard to avoid in the world of Python testing.[1]

1 Although not impossible! Check out the book *Cosmic Python*, which has tips on testing without mocks. I happen to know that at least one of the two authors is incredibly wise.

We'll talk about test fixtures and server-side debugging, and how to set up a continuous integration (CI) environment. None of these things are take-it-or-leave-it, optional, luxury extras for your project—they're all vital!

Inevitably, the learning curve does get a little steeper in this section. You may find yourself having to read things a couple of times before they sink in, or you may find that things don't work on the first go, and that you need to do a bit of debugging on your own.

But I encourage you to persist with it! The harder it is, the more rewarding it is, right? And, remember, I'm always happy to help if you're stuck; just drop me an email at *obeythetestinggoat@gmail.com*.

Come on; I promise the best is yet to come!

CHAPTER 17

A Gentle Excursion into JavaScript

> You can never understand one language until you understand at least two.
>
> —Geoffrey Willans, English author and journalist

Our new validation logic is good, but wouldn't it be nice if the duplicate-item error messages disappeared once the user started fixing the problem, just like our nice HTML5 validation errors do?

Try it—spin up the site with `./src/manage.py runserver`, start a list, and if you try to submit an empty item, you get the "Please fill out this field" pop-up, and it disappears as soon as you enter some text. By contrast, enter an item twice, you get the "You've already got this in your list" message in red—and even if you edit your submission to something valid, the error stays there until you submit the form (see Figure 17-1).

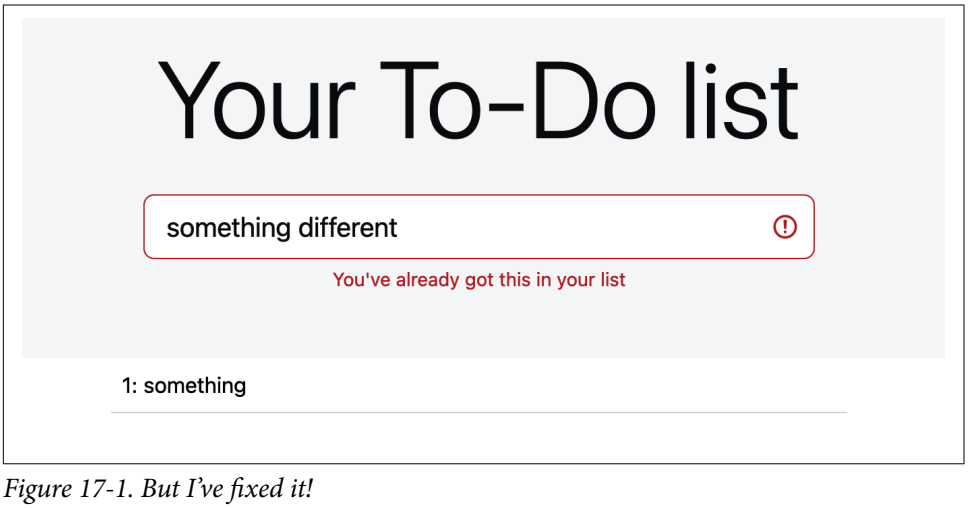


Figure 17-1. But I've fixed it!

To get that error to disappear dynamically, we'd need a teeny-tiny bit of JavaScript. Python is a delightful language to program in. JavaScript wasn't always that. But many of the rough edges have been smoothed off, and I think it's fair to say that JavaScript is actually quite nice now. And in the world of web development, using JavaScript is unavoidable. So let's dip our toes in, and see if we can't have a bit of fun.

I'm going to assume you know the basics of JavaScript syntax. If not, the Mozilla guides on MDN are always good quality. I've also heard good things about *Eloquent JavaScript*, if you prefer a real book.

Starting with an FT

Let's add a new functional test (FT) to the `ItemValidationTest` class; that asserts that our error message disappears when we start typing:

src/functional_tests/test_list_item_validation.py (ch17l001)

```
def test_error_messages_are_cleared_on_input(self):
    # Edith starts a list and causes a validation error:
    self.browser.get(self.live_server_url)
    self.get_item_input_box().send_keys("Banter too thick")
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for_row_in_list_table("1: Banter too thick")
    self.get_item_input_box().send_keys("Banter too thick")
    self.get_item_input_box().send_keys(Keys.ENTER)
    self.wait_for(  ❶
        lambda: self.assertTrue(  ❶
            self.browser.find_element(
                By.CSS_SELECTOR, ".invalid-feedback"
            ).is_displayed()  ❷
        )
    )

    # She starts typing in the input box to clear the error
    self.get_item_input_box().send_keys("a")

    # She is pleased to see that the error message disappears
    self.wait_for(
        lambda: self.assertFalse(
            self.browser.find_element(
                By.CSS_SELECTOR, ".invalid-feedback"
            ).is_displayed()  ❷
        )
    )
```

❶ We use another of our `wait_for` invocations, this time with `assertTrue`.

❷ `is_displayed()` tells you whether an element is visible or not. We can't just rely on checking whether the element is *present* in the DOM, because we're now going to mark elements as hidden, rather than removing them from the document object model (DOM) altogether.

The FT fails appropriately:

```
$ python src/manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input

FAIL: test_error_messages_are_cleared_on_input (functional_tests.test_list_item
_validation.ItemValidationTest.test_error_messages_are_cleared_on_input)
[...]
  File "...goat-book/src/functional_tests/test_list_item_validation.py", line
89, in <lambda>
    lambda: self.assertFalse(
            ~~~~~~~~~~~~~~~~^
        self.browser.find_element(
        ^^^^^^^^^^^^^^^^^^^^^^^^^^
            By.CSS_SELECTOR, ".invalid-feedback"
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        ).is_displayed()
        ^^^^^^^^^^^^^^^^
    )
    ^
AssertionError: True is not false
```

But, before we move on: three strikes and refactor! We've got several places where we find the error element using CSS. Let's move the logic to a helper function:

src/functional_tests/test_list_item_validation.py (ch17l002)

```
class ItemValidationTest(FunctionalTest):
    def get_error_element(self):
        return self.browser.find_element(By.CSS_SELECTOR, ".invalid-feedback")

    [...]
```

And we then make three replacements in *test_list_item_validation*, like this:

src/functional_tests/test_list_item_validation.py (ch17l003)

```
        self.wait_for(
            lambda: self.assertEqual(
                self.get_error_element().text,
                "You've already got this in your list",
            )
        )
[...]
        self.wait_for(
            lambda: self.assertTrue(self.get_error_element().is_displayed()),
        )
[...]
        self.wait_for(
            lambda: self.assertFalse(self.get_error_element().is_displayed()),
        )
```

We still have our expected failure:

```
$ python src/manage.py test functional_tests.test_list_item_validation
[...]
    lambda: self.assertFalse(self.get_error_element().is_displayed()),
            ~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: True is not false
```

I like to keep helper methods in the FT class that's using them, and only promote them to the base class when they're actually needed elsewhere. It stops the base class from getting too cluttered. You ain't gonna need it (YAGNI)!

A Quick Spike

This will be our first bit of JavaScript. We're also interacting with the Bootstrap CSS framework, which we maybe don't know very well.

In Chapter 15, we saw that you can use a unit test as a way of exploring a new API or tool. Sometimes though, you just want to hack something together without any tests at all, just to see if it works, to learn it or get a feel for it.

That's absolutely fine! When learning a new tool or exploring a new possible solution, it's often appropriate to leave the rigorous TDD process to one side, and build a little prototype without tests, or perhaps with very few tests. The Goat doesn't mind looking the other way for a bit.

It's actually *fine* to code without tests sometimes, when you want to explore a new tool or build a throwaway proof-of-concept—as long as you geniunely do throw that hacky code away, and start again with TDD for the real thing. The code *always* comes out much nicer the second time around.

This kind of prototyping activity is often called a "spike", for reasons that aren't entirely clear, but it's a nice memorable name.[1]

Before we start, let's commit our FT. When embarking on a spike, you want to be able to get back to a clean slate:

```
$ git diff  # new method in src/tests/functional_tests/test_list_item_validation.py
$ *git commit -am"FT that validation errors disapper on type"
```

Always do a commit before embarking on a spike.

A Simple Inline Script

I hacked around for a bit, and here's more or less the first thing I came up with. I'm adding the JavaScript inline, in a `<script>` tag at the bottom of our *base.html* template:

src/lists/templates/base.html (ch17l004)

```
    [...]
    </div>

    <script>
      const textInput = document.querySelector("#id_text");  ❶
      textInput.oninput = () => {  ❷❸
        const errorMsg = document.querySelector(".invalid-feedback");
        errorMsg.style.display = "none";  ❹
      }
    </script>

  </body>
</html>
```

1 This chapter shows a very small spike. We'll come back and look at the spiking process again, with a weightier Python/Django example, in Chapter 19 .

❶ `document.querySelector` is a way of finding an element in the DOM, using CSS selector syntax, very much like the Selenium `find_element(By.CSS_SELECTOR)` method from our FTs. Grizzled readers may remember having to use jQuery's `$` function for this.

❷ `oninput` is how you attach an event listener "callback" function, which will be called whenever the user inputs something into the text box.

❸ Arrow functions, `() => {...}`, are the new way of writing anonymous functions in JavaScript, a bit like Python's `lambda` syntax. I think they're cute! Arguments go in the round brackets and the function body goes in the curly brackets. This is a function that takes no arguments—or I should say, ignores any arguments you try to give it. So, what does it do?

❹ It finds the error message element, and then hides it by setting its `style.display` to "none".

That's actually good enough to get our FT passing:

```
$ python src/manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input
Found 1 test(s).
[...]
.
 ----------------------------------------------------------------------
Ran 1 test in 3.284s

OK
```

It's good practice to put your script loads at the end of your body HTML, as it means the user doesn't have to wait for all your JavaScript to load before they can see something on the page. It also helps to make sure most of the DOM has loaded before any scripts run. See also "Columbo Says: Wait for Onload" on page 430 later in this chapter.

Using the Browser DevTools

The test might be happy, but our solution is a little unsatisfactory. If you actually try it in your browser, you'll see that although the error message is gone, the input is still red and invalid-looking (see Figure 17-2).

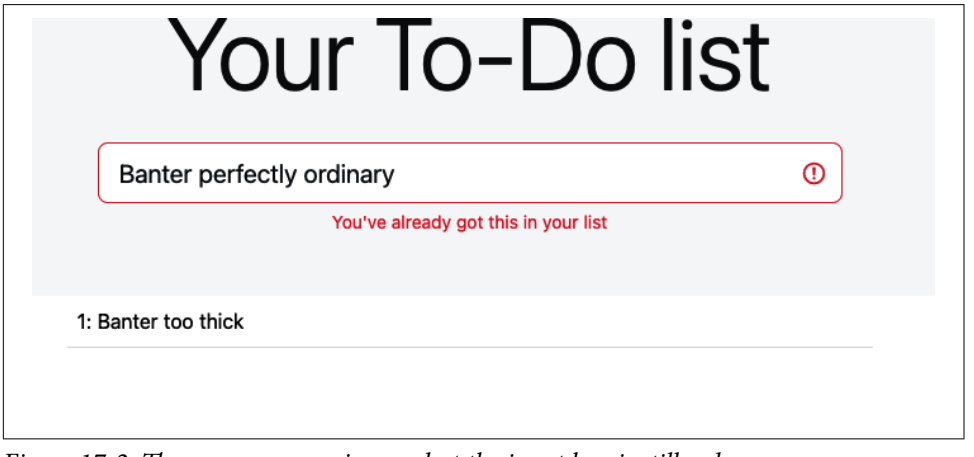


Figure 17-2. The error message is gone but the input box is still red

You're probably imagining that this has something to do with Bootstrap. We might have been able to hide the error message, but we also need to tell Bootstrap that this input no longer has invalid contents.

This is where I'd normally open up DevTools. If level one of hacking is spiking code directly into an inline `<script>` tag, level two is hacking things directly in the browser, where it's not even saved to a file!

In Figure 17-3, you can see me directly editing the HTML of the page, and finding out that removing the `is-invalid` class from the input element seems to do the trick. It not only removes the error message, but also the red border around the input box.

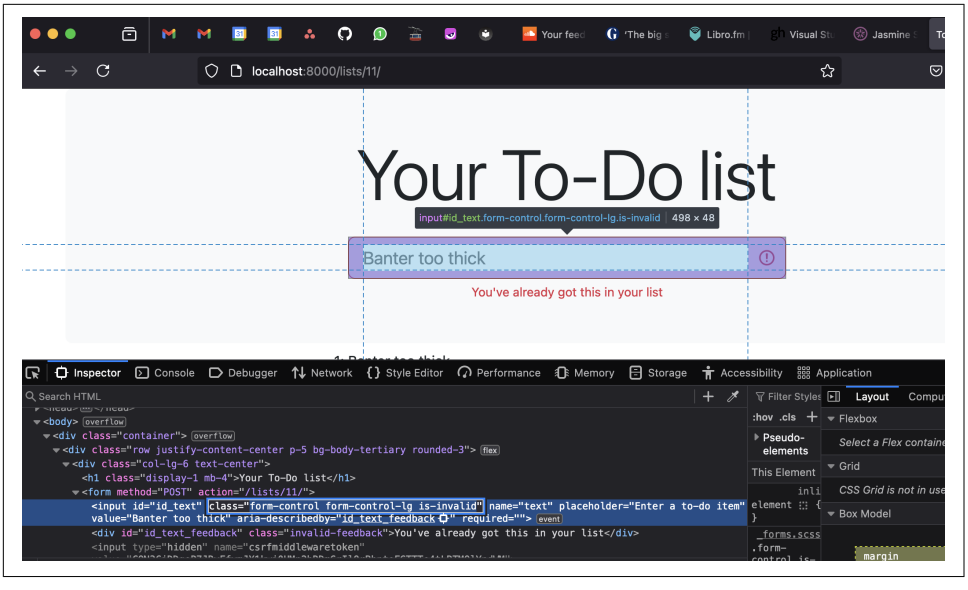


Figure 17-3. Editing the HTML in the browser DevTools

We have a reasonable solution now; let's write it down:

Time to de-spike!

Do We Really Need to Write Unit Tests for This?

Do we really need to write unit tests for this? By this point in the book, you probably know I'm going to say "yes", but let's talk about it anyway.

Our FT definitely covers the functionality that our JavaScript is delivering, and we could extend it if we wanted to, to check on the colour of the input box or to look at the input element's CSS classes. And if I was really sure that this was the *only* bit of JavaScript we were ever going to write, I probably would be tempted to leave it at that.

But I want to press on for two reasons. Firstly, because any book on web development has to talk about JavaScript and, in a TDD book, I have to show a bit of TDD in JavaScript.

More importantly though, as always, we have the boiled frog problem.[2] We might not have enough JavaScript *yet* to justify a full test suite, but what about when we come along later and add a tiny bit more? And a tiny bit more again?

It's always a judgement call. On the one hand YAGNI, but on the other hand, I think it's best to put the scaffolding in place early so that going test-first is the easy choice later.

I can already think of several extra things I'd want to do in the frontend! What about resetting the input to being invalid if someone types in the exact duplicate text again?

Choosing a Basic JavaScript Test Runner

Choosing your testing tools in the Python world is fairly straightforward. The standard library `unittest` package is perfectly adequate, and the Django test runner also makes a good default choice. More and more though, people will choose pytest for its `assert`-based assertions, and its fixture management. We don't need to get into the pros and cons now! The point is that there's a "good enough" default, and there's one main popular alternative.

The JavaScript world has more of a proliferation! Mocha, Karma, Jester, Chai, AVA, and Tape are just a few of the options I came across when researching for the third edition.

I chose Jasmine, because it's still popular despite being around for nearly a decade, and because it offers a "stand-alone" test runner that you can use without needing to dive into the whole Node.js/NPM ecosystem.

2 For a reminder, read back on this problem in "On the Merits of Trivial Tests for Trivial Functions" on page 47.

An Overview of Jasmine

By now, we're used to the way that testing works with Python's `unittest` library:

1. We have a tests file, separate from the code we're actually testing.
2. We have a way of grouping blocks of code into a test: it's a method, whose name starts with `test_`, on a class that inherits from `unittest.TestCase`.
3. We have a way of making assertions in the test (the special `assert` methods, e.g., `self.assertEqual()`).
4. We have a way of grouping related tests together (putting them in the same class).
5. We can specify shared setup and cleanup code that runs before and after all the tests in a given group, the `setUp()` and `tearDown()` methods.
6. We have some additional helpers that set up our app in a way that simulates what happens "in real life"—whether that's Selenium and the `LiveServerTestCase`, or the Django test client. This is sometimes called the "test harness".

There are going to be fairly straightforward equivalents for the first five of these concepts in Jasmine:

1. There is a tests file (*Spec.js*).
2. Tests go into an anonymous function inside an `it()` block.
3. Assertions use a special function called `expect()`, with a syntax based on method chaining for asserting equality.
4. Blocks of related tests go into a function in a `describe()` block.
5. `setUp()` and `tearDown()` are called `beforeEach()` and `afterEach()`, respectively.

There are some differences for sure, but you'll see over the course of the chapter that they're fundamentally the same. What *is* substantially different is the "test harness" part—the way that Jasmine creates an environment for us to work against.

Because we're using the browser runner, what we're actually going to do is define an HTML file (*SpecRunner.html*), and the engine for running our code is going to be an actual browser (with JavaScript running inside it).

That HTML will be the entry point for our tests, so it will be in charge of importing our framework, our tests file, and the code under test. It's essentially a parallel, standalone web page that isn't actually part of our app, but it *does* import the same JavaScript source code that our app uses.

Setting Up Our JavaScript Test Environment

Let's download Jasmine now:

```
$ wget -O jasmine.zip \
  https://github.com/jasmine/jasmine/releases/download/v4.6.1/jasmine-standalone-4.6.1.zip
$ unzip jasmine.zip -d src/lists/static/tests
$ rm jasmine.zip
# if you're on Windows you may not have wget or unzip,
# but i'm sure you can manage to manually download and unzip the jasmine release

# move the example tests "Spec" file to a more central location
$ mv src/lists/static/tests/spec/PlayerSpec.js src/lists/static/tests/Spec.js

# delete all the other stuff we don't need
$ rm -rf src/lists/static/tests/src
$ rm -rf src/lists/static/tests/spec
```

That leaves us with a directory structure like this:

```
$ tree src/lists/static/tests
src/lists/static/tests
├── MIT.LICENSE
├── Spec.js
├── SpecRunner.html
└── lib
    └── jasmine-4.6.1
        ├── boot0.js
        ├── boot1.js
        ├── jasmine-html.js
        ├── jasmine.css
        ├── jasmine.js
        └── jasmine_favicon.png

3 directories, 9 files
```

SpecRunner.html is the file that ties the proverbial room together. So, we need to go edit it to make sure it's pointing at the right places, to take into account the things we've moved around:

```
@@ -14,12 +14,10 @@
    <script src="lib/jasmine-4.6.1/boot1.js"></script>

    <!-- include source files here... -->
-   <script src="src/Player.js"></script>
-   <script src="src/Song.js"></script>
+   <script src="../lists.js"></script>

    <!-- include spec files here... -->
-   <script src="spec/SpecHelper.js"></script>
-   <script src="spec/PlayerSpec.js"></script>
+   <script src="Spec.js"></script>

  </head>
```

We change the source files to point at a (for-now imaginary) *lists.js* file that we'll put into the *static* folder, and we change the spec files to point at the single *Spec.js* file, in the *static/tests* folder.

Our First Smoke Test: Describe, It, Expect

Now, let's open up that *Spec.js* file and strip it down to a single minimal smoke test:

src/lists/static/tests/Spec.js (ch17l007)

```
describe("Superlists JavaScript", () => {  ❶

  it("should have working maths", () => {  ❷
    expect(1 + 1).toEqual(2);  ❸
  });

});
```

❶ The `describe` block is a way of grouping tests together, a bit like we use classes in our Python tests. It starts with a string name, and then an arrow function for its body.

❷ The `it` block is a single test, a bit like a method in a Python test class. Similarly to the `describe` block, we have a name and then a function to contain the test code. As you can see, the convention is for the descriptive name to complete the sentence started by `it`, in the context of the `describe()` block earlier; so, they often start with "should".

❸ Now we have our assertion. This is a little different from assertions in unittest; it's using what's sometimes called "expect" style, often also seen in the Ruby world. We wrap our "actual" value in the `expect()` function, and then our assertions are methods on the resulting expect object, where `.toEqual` is the equivalent of `assertEqual` in Python.

Running the Tests via the Browser

Let's see how that looks. Open up *SpecRunner.html* in your browser; you can do this from the command line with:

```
$ firefox src/lists/static/tests/SpecRunner.html
# or, on a mac:
$ open src/lists/static/tests/SpecRunner.html
```

Or, you can navigate to it in the address bar, using the `file://` protocol—something like this: *file://home/your-username/path/to/superlists/src/lists/static/tests/SpecRunner.html*.

Either way you get there, you should see something like Figure 17-4.

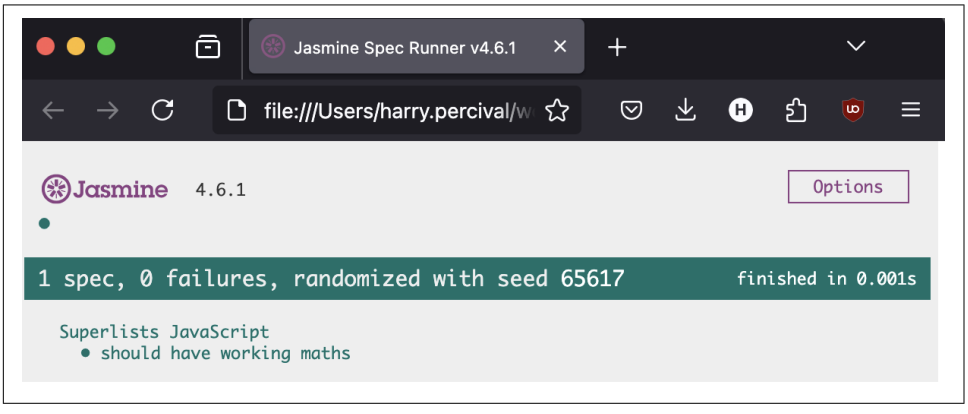


Figure 17-4. The Jasmine spec runner in action

Let's try adding a deliberate failure to see what that looks like:

src/lists/static/tests/Spec.js (ch17l008)

```
it("should have working maths", () => {
  expect(1 + 1).toEqual(3);
});
```

Now if we refresh our browser, we'll see red (Figure 17-5).

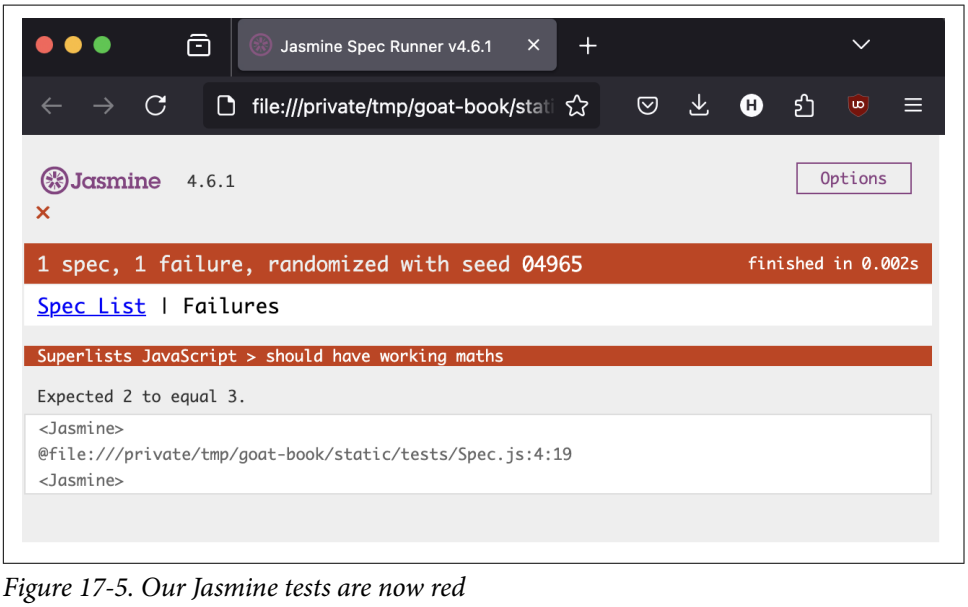


Figure 17-5. Our Jasmine tests are now red

Is the Jasmine Standalone Browser Test Runner Unconventional?

Is the Jasmine standalone browser test runner unconventional? I think it probably is, to be honest. Although, the JavaScript world moves so fast, so I could be wrong by the time you read this.

What I do know is that, along with moving very fast, JavaScript things can very quickly become very complicated. A lot of people are working with frameworks these days (React being the main one), and that comes with TypeScript, transpilers, Node.js, NPM, the massive *node_modules* folder—and a very steep learning curve.

In this chapter, my aim is to stick with the basics. The standalone/browser-based test runner lets us write tests without needing to install Node.js or anything else, and it lets us test interactions with the DOM. That's enough to give us a basic environment in which to do TDD in JavaScript.

If you decide to go further in the world of frontend, you probably will eventually get into the complexity of frameworks and TypeScript and transpilers, but the basics we work with here will still be a good foundation.

We will actually take things a small step further in this book, including dipping our toes into NPM and Node.js in Chapter 25, where we *will* get CLI-based JavaScript tests working. So, look out for that!

Testing with Some DOM Content

What do we *actually* want to test? We want some JavaScript that will hide the `.invalid-feedback` error div when the user starts typing into the input box. In other words, our code is going to interact with the `input` element on the page and with the `div.invalid-feedback`.

Let's look at how to set up some copies of these elements in our JavaScript test environment, for our tests and our code to interact with:

src/lists/static/tests/Spec.js (ch17l010)

```
describe("Superlists JavaScript", () => {
  let testDiv;  ❹

  beforeEach(() => {  ❶
    testDiv = document.createElement("div");  ❷
    testDiv.innerHTML = `  ❸
      <form>
        <input
          id="id_text"
          name="text"
          class="form-control form-control-lg is-invalid"
          placeholder="Enter a to-do item"
          value="Value as submitted"
          aria-describedby="id_text_feedback"
          required
        />
        <div id="id_text_feedback" class="invalid-feedback">An error message</div>
      </form>
    `;
    document.body.appendChild(testDiv);
  });

  afterEach(() => {  ❶
    testDiv.remove();
  });
```

❶ The `beforeEach` and `afterEach` functions are Jasmine's equivalent of `setUp` and `tearDown`.

❷ The `document` global is a built-in browser variable that represents the current HTML page. So, in our case, it's a reference to the *SpecRunner.html* page.

❸ We create a new `div` element and populate it with some HTML that matches the elements we care about from our Django template. Notice the use of backticks (`` ` ``) to enable us to write multiline strings. Depending on your text editor, it may even nicely syntax-highlight the HTML for you.

❹ A little quirk of JavaScript here, because we want the same `testDiv` variable to be available inside both the `beforeEach` and `afterEach` functions: we declare the variable with `let` in the containing scope outside of both functions.

In theory, we could have just added the HTML to the *SpecRunner.html* file, but by using `beforeEach` and `afterEach`, I'm making sure that each test gets a completely fresh copy of the HTML elements involved, so that one test can't affect another.

To ensure isolation between browser-based JavaScript tests, use `beforeEach()` and `afterEach()` to create and tidy up any DOM elements that your code needs to interact with.

Let's now play with our testing framework to see if we can find DOM elements and make assertions on whether they are visible. We'll also try the same `style.display=none` hiding technique that we originally used in our spiked code:

src/lists/static/tests/Spec.js (ch17l011)

```
it("should have a useful html fixture", () => {
  const errorMsg = document.querySelector(".invalid-feedback");
  expect(errorMsg.checkVisibility()).toBe(true);  ❶
});

it("can hide things manually and check visibility in tests", () => {
  const errorMsg = document.querySelector(".invalid-feedback");
  errorMsg.style.display = "none";  ❷
  expect(errorMsg.checkVisibility()).toBe(false);  ❸
});
```

❶ We retrieve our error `div` with `querySelector` again, and then use another fairly new API in JavaScript-Land called `checkVisibility()` to check if it's displayed or hidden.[3]

❷ We *manually* hide the element in the test, by setting its `style.display` to "none". (Again, our objective here is to smoke-test, both our ability to hide things and our ability to test that they are hidden.)

❸ And we check it worked, with `checkVisibility()` again.

Notice that I'm being really good about splitting things out into multiple tests, with one assertion each. Jasmine encourages that by deprecating the ability to pass failure messages into individual `expect/toBe` expressions, for example.

3 Read up on the `checkVisibility()` method in the MDN documentation.

If you refresh the browser, you should see that all passes:

```
2 specs, 0 failures, randomized with seed 12345      finished in 0.005s


Superlists JavaScript
  * can hide things manually and check visibility in tests
  * should have a useful html fixture
```

(From now on, I'll show the Jasmine outputs as text, like this, to avoid filling the chapter with screenshots.)

Building a JavaScript Unit Test for Our Desired Functionality

Now that we're acquainted with our JavaScript testing tools, we can start to write the real thing:

src/lists/static/tests/Spec.js (ch17l012)

```
it("should have a useful html fixture", () => {  ❶
  const errorMsg = document.querySelector(".invalid-feedback");
  expect(errorMsg.checkVisibility()).toBe(true);
});

it("should hide error message on input", () => {  ❷
  const textInput = document.querySelector("#id_text");  ❸
  const errorMsg = document.querySelector(".invalid-feedback");

  textInput.dispatchEvent(new InputEvent("input"));  ❹

  expect(errorMsg.checkVisibility()).toBe(false);  ❺
});
```

❶ As it's not doing any harm, let's keep the first smoke test.

❷ Let's change the second one, and give it a name that describes what we want to happen; our objective is that, when the user starts typing into the input box, we should hide the error message.

❸ We retrieve the `<input>` element from the DOM, in a similar way to how we found the error message `div`.

❹ Here's how we simulate a user typing into the input box.

❺ And here's our real assertion: the error `div` should be hidden after the input box sees an input event.

That gives us our expected failure:

```
2 specs, 1 failure, randomized with seed 12345      finished in 0.005s

Spec List | Failures

Superlists JavaScript > should hide error message on input
Expected true to be false.
<Jasmine>
@file:///...goat-book/src/lists/static/tests/Spec.js:38:40
<Jasmine>
```

Now let's try reintroducing the code we hacked together in our spike, into *lists.js*:

src/lists/static/lists.js (ch17l014)

```
const textInput = document.querySelector("#id_text");
textInput.oninput = () => {
  const errorMsg = document.querySelector(".invalid-feedback");
  errorMsg.style.display = "none";
};
```

That doesn't work! We get an unexpected error:

```
2 specs, 2 failures, randomized with seed 12345      finished in 0.005s
Error during loading: TypeError: textInput is null in
file:///...goat-book/src/lists/static/lists.js line 2
Spec List | Failures

Superlists JavaScript > should hide error message on input
Expected true to be false.
<Jasmine>
@file:///...goat-book/src/lists/static/tests/Spec.js:38:40
<Jasmine>
```

If your Jasmine output shows `Script error` instead of `textInput is null`, open up the DevTools console, and you'll see the actual error printed in there, as in Figure 17-6.[4]

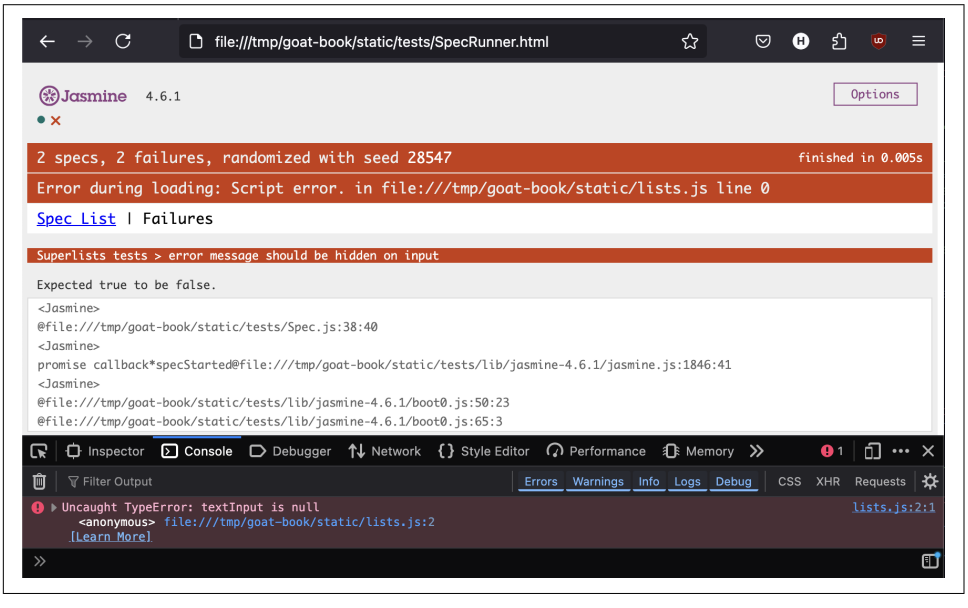


Figure 17-6. `textInput` is null, one way or another

`textInput is null`, it says. Let's see if we can figure out why.

Fixtures, Execution Order, and Global State: Key Challenges of JavaScript Testing

One of the difficulties with JavaScript in general, and testing in particular, is understanding the order of execution of our code (i.e., what happens when). When does our code in *lists.js* run, and when do each of our tests run? How do they all interact with global state—that is, the DOM of our web page and the fixtures that we've already seen are supposed to be cleaned up after each test?

4 Some users have also reported that Google Chrome will show a different error, to do with the browser preventing loading local files. If you really can't use Firefox, you might be able to find some solutions on Stack Overflow.

console.log for Debug Printing

Let's add a couple of debug prints, or "console.logs":

src/lists/static/tests/Spec.js (ch17l015)

```
console.log("Spec.js loading");

describe("Superlists JavaScript", () => {
  let testDiv;

  beforeEach(() => {
    console.log("beforeEach");
    testDiv = document.createElement("div");

    [...]

  it("should have a useful html fixture", () => {
    console.log("in test 1");
    const errorMsg = document.querySelector(".invalid-feedback");
    [...]

  it("should hide error message on input", () => {
    console.log("in test 2");
    const textInput = document.querySelector("#id_text");
    [...]
```

And the same in our actual JavaScript code:

src/lists/static/lists.js (ch17l016)

```
console.log("lists.js loading");
const textInput = document.querySelector("#id_text");
textInput.oninput = () => {
  const errorMsg = document.querySelector(".invalid-feedback");
  errorMsg.style.display = "none";
};
```

Rerun the tests, opening up the browser debug console (Ctrl+Shift+I or Cmd+Alt+I) and you should see something like Figure 17-7.

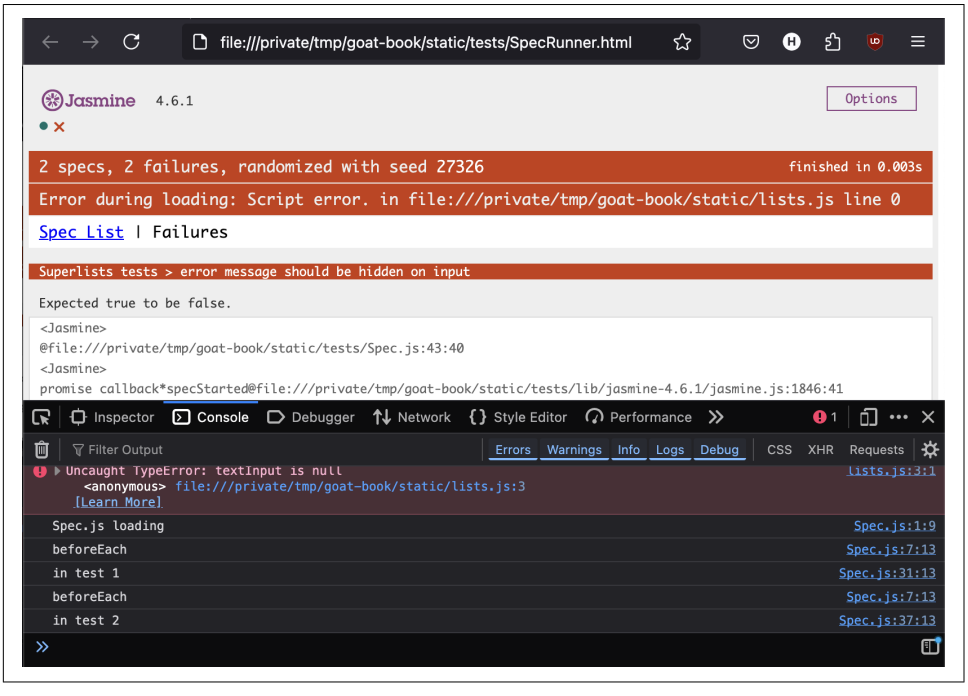


Figure 17-7. Jasmine tests with `console.log` debug outputs

What do we see?

1. First, *lists.js* loads.
2. Then, we see the error saying `textInput is null`.
3. Next, we see our tests loading in *Spec.js*.
4. Then, we see a `beforeEach`, which is when our test fixture actually gets added to the DOM.
5. Finally, we see the first test run.

This explains the problem: when *lists.js* loads, the input node doesn't exist yet.

Using an Initialize Function for More Control Over Execution Time

We need more control over the order of execution of our JavaScript. Rather than just relying on the code in *lists.js* running whenever it is loaded by a `<script>` tag, we can use a common pattern: define an "initialize" function and call that when we want to in our tests (and later in real life).[5]

Here's what that function could look like:

src/lists/static/lists.js (ch17l017)

```
console.log("lists.js loading");
const initialize = () => {
  console.log("initialize called");
  const textInput = document.querySelector("#id_text");
  textInput.oninput = () => {
    const errorMsg = document.querySelector(".invalid-feedback");
    errorMsg.style.display = "none";
  };
};
```

And in our tests file, we call `initialize()` in our key test:

src/lists/static/tests/Spec.js (ch17l018)

```
  it("should have a useful html fixture", () => {
    console.log("in test 1");
    const errorMsg = document.querySelector(".invalid-feedback");
    expect(errorMsg.checkVisibility()).toBe(true);
  });

  it("should hide error message on input", () => {
    console.log("in test 2");
    const textInput = document.querySelector("#id_text");
    const errorMsg = document.querySelector(".invalid-feedback");

    initialize();  ❶
    textInput.dispatchEvent(new InputEvent("input"));

    expect(errorMsg.checkVisibility()).toBe(false);
  });
});
```

5 Have you been enjoying the British English spelling in the book so far and are shocked to see the *z* in "initialize"? By convention, even us Brits often use American spelling in code, because it makes it easier for international colleagues to read, and to make it correspond better with code samples on the internet.

❶ This is where we call `initialize()`. We don't need to call it in our fixture sense-check.

And that will actually get our tests passing!

```
2 specs, 0 failures, randomized with seed 12345      finished in 0.005s


Superlists JavaScript
  * should hide error message on input
  * should have a useful html fixture
```

And now the `console.log` outputs should be in a more sensible order:

```
lists.js loading            lists.js:1:9
Spec.js loading             Spec.js:1:9
beforeEach                  Spec.js:7:13
in test 1                   Spec.js:31:13
beforeEach                  Spec.js:7:13
in test 2                   Spec.js:37:13
initialize called           lists.js:3:11
```

Deliberately Breaking Our Code to Force Ourselves to Write More Tests

I'm always nervous when I see green tests. We've copy-pasted five lines of code from our spike with just one test. That was a little too easy, even if we did have to go through that little `initialize()` dance.

So, let's change our `initialize()` function to deliberately break it. What if we just immediately hide errors?

src/lists/static/lists.js (ch17l019)

```
const initialize = () => {
  // const textInput = document.querySelector("#id_text");
  // textInput.oninput = () => {
    const errorMsg = document.querySelector(".invalid-feedback");
    errorMsg.style.display = "none";
  // };
};
```

Oh dear, as I feared—the tests just pass:

```
2 specs, 0 failures, randomized with seed 12345      finished in 0.005s


Superlists JavaScript
  * should hide error message on input
  * should have a useful html fixture
```

We need an extra test, to check that our `initialize()` function isn't overzealous:

src/lists/static/tests/Spec.js (ch17l020)

```
it("should hide error message on input", () => {
  [...]
});

it("should not hide error message before event is fired", () => {
  const errorMsg = document.querySelector(".invalid-feedback");
  initialize();
  expect(errorMsg.checkVisibility()).toBe(true);  ❶
});
```

❶ In this test, we don't fire the input event with `dispatchEvent`, so we expect the error message to still be visible.

That gives us our expected failure:

```
3 specs, 1 failure, randomized with seed 12345      finished in 0.005s

Spec List | Failures

Superlists JavaScript > should not hide error message before event is fired
Expected false to be true.
<Jasmine>
@file:///...goat-book/src/lists/static/tests/Spec.js:48:40
<Jasmine>
```

This justifies us to restore the `textInput.oninput()`:

src/lists/static/lists.js (ch17l021)

```
const initialize = () => {
  const textInput = document.querySelector("#id_text");
  textInput.oninput = () => {
    const errorMsg = document.querySelector(".invalid-feedback");
    errorMsg.style.display = "none";
  };
};
```

Red/Green/Refactor: Removing Hardcoded Selectors

The `#id_text` and `.invalid-feedback` selectors are "magic constants" at the moment. It would be better to pass them into `initialize()`, both in the tests and in *base.html*, so that they're defined in the same file that actually has the HTML elements.

And while we're at it, our tests could do with a bit of refactoring too, to remove some duplication. We'll start with that, by defining a few more variables in the top-level scope, and populate them in the `beforeEach`:

src/lists/static/tests/Spec.js (ch17l022)

```
describe("Superlists JavaScript", () => {
  const inputId = "id_text";  ❶
  const errorClass = "invalid-feedback";  ❶
  const inputSelector = `#${inputId}`;  ❷
  const errorSelector = `.${errorClass}`;  ❷
  let testDiv;
  let textInput;  ❸
  let errorMsg;  ❸

  beforeEach(() => {
    console.log("beforeEach");
    testDiv = document.createElement("div");
    testDiv.innerHTML = `
      <form>
        <input
          id="${inputId}"  ❹
          name="text"
          class="form-control form-control-lg is-invalid"
          placeholder="Enter a to-do item"
          value="Value as submitted"
          aria-describedby="id_text_feedback"
          required
        />
        <div id="id_text_feedback" class="${errorClass}">An error message</div>  ❹
      </form>
    `;
    document.body.appendChild(testDiv);
    textInput = document.querySelector(inputSelector);  ❺
    errorMsg = document.querySelector(errorSelector);  ❺
  });
```

❶ Let's define some constants to represent the selectors for our input element and our error message `div`.

❷ We can use JavaScript's string interpolation (the equivalent of f-strings) to then define the CSS selectors for the same elements.

❸ We'll also set up some variables to hold the elements we're always referring to in our tests (these can't be constants, as we'll see shortly).

❹ We use a bit more interpolation to reuse the constants in our HTML template. A first bit of de-duplication!

❺ Here's why `textInput` and `errorMsg` can't be constants: we're re-creating the DOM fixture in every `beforeEach`, so we need to re-fetch the elements each time.

Now we can apply some DRY ("don't repeat yourself") to strip down our tests:

src/lists/static/tests/Spec.js (ch17l023)

```
it("should have a useful html fixture", () => {
  expect(errorMsg.checkVisibility()).toBe(true);
});

it("should hide error message on input", () => {
  initialize();
  textInput.dispatchEvent(new InputEvent("input"));

  expect(errorMsg.checkVisibility()).toBe(false);
});

it("should not hide error message before event is fired", () => {
  initialize();
  expect(errorMsg.checkVisibility()).toBe(true);
});
```

You can definitely overdo DRY in test, but I think this is working out very nicely. Each test is between one and three lines long, meaning it's very easy to see what each one is doing, and what it's doing differently from the others.

We've only refactored the tests so far, so let's check that they still pass:

```
3 specs, 0 failures, randomized with seed 12345      finished in 0.005s


Superlists JavaScript
  * should hide error message on input
  * should have a useful html fixture
  * should not hide error message before event is fired
```

The next refactor is wanting to pass the selectors to `initialize()`. Let's see what happens if we just do that straight away, in the tests:

src/lists/static/tests/Spec.js (ch17l024)

```
@@ -40,14 +40,14 @@ describe("Superlists JavaScript", () => {
   });

   it("should hide error message on input", () => {
-    initialize();
+    initialize(inputSelector, errorSelector);
     textInput.dispatchEvent(new InputEvent("input"));

     expect(errorMsg.checkVisibility()).toBe(false);
   });

   it("should not hide error message before event is fired", () => {
-    initialize();
+    initialize(inputSelector, errorSelector);
     expect(errorMsg.checkVisibility()).toBe(true);
   });
 });
```

Now we look at the tests:

```
3 specs, 0 failures, randomized with seed 12345      finished in 0.005s


Superlists JavaScript
  * should hide error message on input
  * should have a useful html fixture
  * should not hide error message before event is fired
```

They still pass!

You might have been expecting a failure to do with the fact that `initialize()` was defined as taking no arguments—but we passed two! That's because JavaScript is too chill for that. You can call a function with too many or too few arguments, and JavaScript will just *deal with it*.

Let's fish those arguments out in `initialize()`:

src/lists/static/lists.js (ch17l025)

```
const initialize = (inputSelector, errorSelector) => {
  const textInput = document.querySelector(inputSelector);
  textInput.oninput = () => {
    const errorMsg = document.querySelector(errorSelector);
    errorMsg.style.display = "none";
  };
};
```

And the tests still pass:

```
3 specs, 0 failures, randomized with seed 12345      finished in 0.005s
```

Let's deliberately use the arguments the wrong way round, just to check we get a failure:

src/lists/static/lists.js (ch17l026)

```
const initialize = (errorSelector, inputSelector) => {
```

Phew, that does indeed fail:

```
3 specs, 1 failure, randomized with seed 12345      finished in 0.005s

Spec List | Failures

Superlists JavaScript > should hide error message on input
Expected true to be false.
<Jasmine>
@file:///...goat-book/src/lists/static/tests/Spec.js:46:40
<Jasmine>
```

OK, back to the right way around:

src/lists/static/lists.js (ch17l027)

```
const initialize = (inputSelector, errorSelector) => {
```

Does it Work?

And for the moment of truth, we'll pull in our script and invoke our initialize function on our real pages. Let's use another `<script>` tag to include our *lists.js*, and strip down the the inline JavaScript to just calling `initialize()` with the right selectors:

src/lists/templates/base.html (ch17l028)

```
      </div>

      <script src="/static/lists.js"></script>
      <script>
        initialize("#id_text", ".invalid-feedback");
      </script>

    </body>
</html>
```

Aaaand we run our FT:

```
$ python src/manage.py test functional_tests.test_list_item_validation.\
ItemValidationTest.test_error_messages_are_cleared_on_input
[...]

Ran 1 test in 3.023s

OK
```

Hooray! That's a commit!

```
$ git add src/lists
$ git commit -m"Despike our js, add jasmine tests"
```

We're using a `<script>` tag to import our code, but modern JavaScript lets you use `import` and `export` to explicitly import particular parts of your code. However, that involves specifying the scripts as modules, which is fiddly to get working with the single-file test runner we're using. So, I decided to use the "simple" old-fashioned way. By all means, investigate modules in your own projects!

Testing Integration with CSS and Bootstrap

As the tests flashed past, you may have noticed an unsatisfactory bit of red, still left around our input box. Wait a minute! We forgot one of the key things we learned in our spike!

We don't need to manually hack `style.display=none`; we can work *with* the Bootstrap framework and just remove the `.is-invalid` class.

OK, let's try it in our implementation:

src/lists/static/lists.js (ch17l029)

```
const initialize = (inputSelector, errorSelector) => {
  const textInput = document.querySelector(inputSelector);
  textInput.oninput = () => {
    textInput.classList.remove("is-invalid");
  };
};
```

Oh dear; it seems like that doesn't quite work:

```
3 specs, 1 failure, randomized with seed 12345      finished in 0.005s

Spec List | Failures

Superlists JavaScript > should hide error message on input
Expected true to be false.
<Jasmine>
@file:///...goat-book/src/lists/static/tests/Spec.js:46:40
<Jasmine>
```

What's happening here? Well, as hinted in the section title, we're now relying on the integration with Bootstrap's CSS, but our test runner doesn't know about Bootstrap yet.

We can include it in a reasonably familiar way, which is by including it in the `<head>` of our *SpecRunner.html* file:

src/lists/static/tests/SpecRunner.html (ch17l030)

```
  <link rel="stylesheet" href="lib/jasmine-4.6.1/jasmine.css">

  <!-- Bootstrap CSS -->
  <link href="../bootstrap/css/bootstrap.min.css" rel="stylesheet">

  <script src="lib/jasmine-4.6.1/jasmine.js"></script>
```

That gets us back to passing tests:

```
3 specs, 0 failures, randomized with seed 12345      finished in 0.005s


Superlists JavaScript
  * should hide error message on input
  * should have a useful html fixture
  * should not hide error message before event is fired
```

Let's do a little more refactoring. If your editor is set up to do some JavaScript linting, you might have seen a warning saying:

```
'errorSelector' is declared but its value is never read.
```

Great! Looks like we can get away with just one argument to our `initialize()` function:

src/lists/static/lists.js (ch17l031)

```
const initialize = (inputSelector) => {
  const textInput = document.querySelector(inputSelector);
  textInput.oninput = () => {
    textInput.classList.remove("is-invalid");
  };
};
```

Are you enjoying the way the tests keep passing even though we're giving the function too many arguments? JavaScript is so chill, man. Let's strip them down anyway:

src/lists/static/tests/Spec.js (ch17l032)

```
@@ -40,14 +40,14 @@ describe("Superlists JavaScript", () => {
   });

   it("should hide error message on input", () => {
-    initialize(inputSelector, errorSelector);
+    initialize(inputSelector);
     textInput.dispatchEvent(new InputEvent("input"));

     expect(errorMsg.checkVisibility()).toBe(false);
   });

   it("should not hide error message before event is fired", () => {
-    initialize(inputSelector, errorSelector);
+    initialize(inputSelector);
     expect(errorMsg.checkVisibility()).toBe(true);
   });
 });
```

And the base template, yay. Nothing more satisfying than *deleting code*:

src/lists/templates/base.html (ch17l033)

```
    <script>
      initialize("#id_text");
    </script>
```

And we can run the FT one more time, just for safety:

```
OK
```

Trade-offs in JavaScript Unit Testing Versus Selenium

Similarly to the way our Selenium tests and our Django unit tests interact, we have an overlap between the functionality covered by our JavaScript unit tests and our Selenium FTs.

As always, the downside of the FTs is that they are slow, and they can't always point you towards exactly what went wrong. But they *do* give us the best reassurance that all our components—in this case, browser, CSS framework, and JavaScript—are all working together.

On the other hand, by using the jasmine-browser-runner, we are *also* testing the integration between our browser, our JavaScript, and Bootstrap. This comes at the expense of having a slightly clunky testing setup.

If you wanted to switch to faster, more focused unit tests, you could try the following:

- Stop using the browser runner.
- Switch to a node-based CLI test runner.
- Change from asserting using `checkVisibility()` (which won't work without a real DOM) to asserting what the JavaScript code is actually doing—removing the `.is-invalid` CSS class.

It might look something like this:

src/lists/static/tests/Spec.js

```
it("should hide error message on input", () => {
  initialize(inputSelector);
  textInput.dispatchEvent(new InputEvent("input"));

  expect(errorMsg.classList).not.toContain("is-invalid");
});
```

The trade-off here is that you get faster, more focused unit tests, but you need to lean more heavily on Selenium to test the integration with Bootstrap. That could be worth it, but probably only if you start to have a lot more JavaScript code.

Columbo Says: Wait for Onload

> Wait, there's just one more thing...
>
> —Columbo (fictional trench-coat-wearing American detective known for his persistence)

As always, there's one final thing. Whenever you have some JavaScript that interacts with the DOM, it's good to wrap it in some "onload" boilerplate to make sure that

the page has fully loaded before it tries to do anything. Currently it works anyway, because we've placed the `<script>` tag right at the bottom of the page, but we shouldn't rely on that.

The MDN documentation on this is good, as usual.

The modern JavaScript onload boilerplate is minimal:

src/lists/templates/base.html (ch17l034)

```
<script>
  window.onload = () => {
    initialize("#id_text");
  };
</script>
```

That's a commit, folks!

```
$ git status
$ git add src/lists/static  # all our js and tests
$ git add src/lists/templates  # changes to the base template
$ git commit -m"Javascript to hide error messages on input"
```

JavaScript Testing in the TDD Cycle

You may be wondering how these JavaScript tests fit in with our "double loop" TDD cycle (see Figure 17-8).

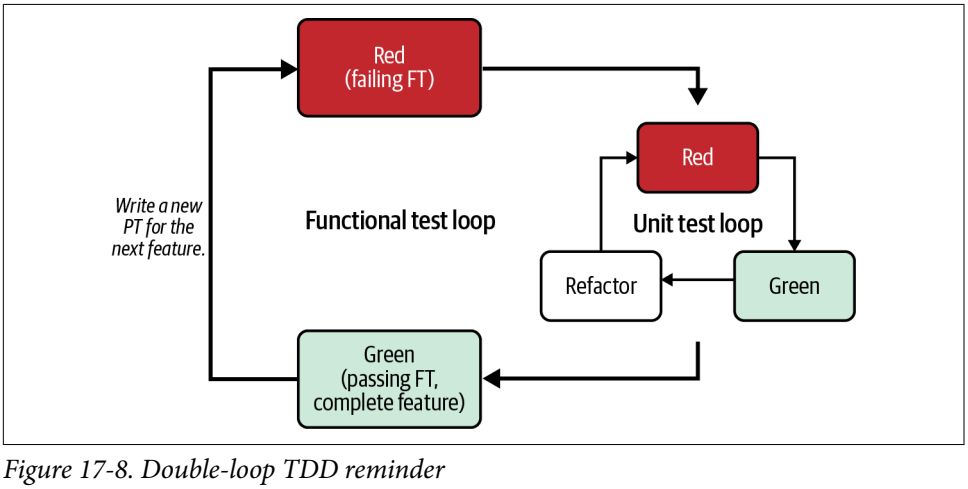


Figure 17-8. Double-loop TDD reminder

The answer is that the JavaScript unit-test/code cycle plays exactly the same role as the Python unit one:

1. Write an FT and see it fail.
2. Figure out what kind of code you need next: Python or JavaScript?
3. Write a unit test in either language, and see it fail.
4. Write some code in either language, and make the test pass.
5. Rinse and repeat.

Phew. Well, hopefully some sense of closure there. The next step is to deploy our new code to our servers.

There is more JavaScript fun in this book too! Have a look at the Online Appendix: Building a REST API), when you're ready for it.

Want a little more practice with JavaScript? See if you can get our error messages to be hidden when the user clicks inside the input element, as well as just when they type in it. You should be able to FT it too, if you want a bit of extra Selenium practice.

JavaScript Testing Notes

Selenium as the outer loop
: One of the great advantages of Selenium is that it enables you to test that your JavaScript really works, just as it tests your Python code. But, as always, FTs are a very blunt tool, so it's often worth pairing them with some lower-level tests.

Choosing your testing framework
: There are many JavaScript test-running libraries out there. Jasmine has been around for a while, but the others are also worth investigating.

Idiosyncrasies of the browser
: No matter which testing library you use, if you're working with Vanilla JavaScript (i.e., not a framework like React), you'll need to work around the key "gotchas" of JavaScript:

  - The DOM and HTML fixtures
  - Global state
  - Understanding and controlling execution order

Frontend frameworks
: An awful lot of frontend work these days is done in frameworks, React being the 1,000-pound gorilla. There are lots of resources on React testing out there, so I'll let you go out and find them if you need them.

CHAPTER 18

Deploying Our New Code

It's time to deploy our brilliant new validation code to our live servers. This will be a chance to see our automated deploy scripts in action for the second time. Let's take the opportunity to make a little deployment checklist.

At this point I always want to say a huge thanks to Andrew Godwin and the whole Django team. In the first edition, I used to have a whole long section, entirely devoted to migrations. Since Django 1.7, migrations now "just work", so I was able to drop it altogether. I mean yes this all happened nearly ten years ago, but still—open source software is a gift. We get such amazing things, entirely for free. It's worth taking a moment to be grateful, now and again.

The Deployment Checklist

Let's make a little checklist of pre-deployment tasks:

1. We run all our unit tests and functional tests (FTs) in the regular way—just in case!
2. We rebuild our Docker image and run our tests against Docker on our local machine.
3. We deploy to staging, and run our FTs against staging.
4. Now we can deploy to prod.

A deployment checklist like this should be a temporary measure. Once you've worked through it manually a few times, you should be looking to take the next step in automation: continuous deployment straight to production using a CI/CD (continuous integration/continuous development) pipeline. We'll touch on this in Chapter 25.

A Full Test Run Locally

Of course, under the watchful eye of the Testing Goat, we're running the tests all the time! But, just in case:

```
$ cd src && python manage.py test
[...]

Ran 37 tests in 15.222s

OK
```

Quick Test Run Against Docker

The next step towards production is running things in Docker. This was one of the main reasons we went to the trouble of containerising our app: to reproduce the production environment as faithfully as possible on our own machine.

So let's rebuild our Docker image and spin up a local Docker container:

```
$ *docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -e DJANGO_DB_PATH=/home/nonroot/db.sqlite3 \
    -it superlists
 => [internal] load build definition from Dockerfile                 0.0s
 => => transferring dockerfile: 371B                                 0.0s
 => [internal] load metadata for docker.io/library/python:3.14-slim   1.4s
 [...]
 => => naming to docker.io/library/superlists                        0.0s
+ docker run -p 8888:8888 --mount
type=bind,source="$PWD/src/db.sqlite3",target=/src/db.sqlite3 -e
DJANGO_SECRET_KEY=sekrit -e DJANGO_ALLOWED_HOST=localhost -e EMAIL_PASSWORD -it
superlists
[2025-01-27 21:29:37 +0000] [7] [INFO] Starting gunicorn 22.0.0
[2025-01-27 21:29:37 +0000] [7] [INFO] Listening at: http://0.0.0.0:8888 (7)
[2025-01-27 21:29:37 +0000] [7] [INFO] Using worker: sync
[2025-01-27 21:29:37 +0000] [8] [INFO] Booting worker with pid: 8
```

And now, in a separate terminal, we can run our FT suite against the Docker:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests
[...]
......
 ----------------------------------------------------------------------
Ran 6 tests in 17.047s

OK
```

Looking good! Let's move on to staging.

Staging Deploy and Test Run

Here's our `ansible-playbook` command to deploy to staging:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -vv
[...]

PLAY [all] *********************************************************************

TASK [Gathering Facts] *********************************************************
[...]
ok: [staging.ottg.co.uk]

TASK [Install docker] **********************************************************
ok: [staging.ottg.co.uk] => {"cache_update_time": [...]

TASK [Add our user to the docker group, so we don't need sudo/become] **********
ok: [staging.ottg.co.uk] => {"append": true, "changed": false, [...]

TASK [Reset ssh connection to allow the user/group change to take effect] ******

TASK [Build container image locally] *******************************************
changed: [staging.ottg.co.uk -> 127.0.0.1] => {"actions": ["Built image
[...]

TASK [Export container image locally] ******************************************
changed: [staging.ottg.co.uk -> 127.0.0.1] => {"actions": ["Archived image [...]

TASK [Upload image to server] **************************************************
changed: [staging.ottg.co.uk] => {"changed": true, "checksum": [...]

TASK [Import container image on server] ****************************************
changed: [staging.ottg.co.uk] => {"actions": ["Loaded image superlists:latest
[...]

TASK [Ensure .secret-key file exists] ******************************************
ok: [staging.ottg.co.uk] => {"changed": false, "dest":
[...]

TASK [Read secret key back from file] ******************************************
ok: [staging.ottg.co.uk] => {"changed": false, "content":
[...]

TASK [Ensure db.sqlite3 file exists outside container] *************************
changed: [staging.ottg.co.uk] => {"changed": true, "dest": [...]

TASK [Run container] ***********************************************************
changed: [staging.ottg.co.uk] => {"changed": true, "container":
[...]

TASK [Run migration inside container] ******************************************
changed: [staging.ottg.co.uk] => {"changed": true, "rc": 0, "stderr": "",
[...]

PLAY RECAP *********************************************************************
staging.ottg.co.uk         : ok=12   changed=7    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
```

If your server is offline because you ran out of free credits with your provider, you'll have to create a new one. Skip back to Chapter 11 if you need.

And now we run the FTs against staging:

```
$ TEST_SERVER=staging.ottg.co.uk python src/manage.py test functional_tests
OK
```

Hooray!

Production Deploy

As all is looking well, we can deploy to prod!

```
$ ansible-playbook --user=elspeth -i www.ottg.co.uk, infra/deploy-playbook.yaml -vv
```

What to Do If You See a Database Error

Because our migrations introduce a new integrity constraint, you may find that it fails to apply because some existing data violates that constraint. For example, here's what you might see if any of the lists on the server already contain duplicate items:

```
sqlite3.IntegrityError: columns list_id, text are not unique
```

At this point, you have two choices:

1. Delete the database on the server and try again—after all, it's only a toy project!
2. Create a data migration. You can find out more in the Django migrations docs.

How to Delete the Database on the Staging Server

Here's how you might do option 1:

```
ssh elspeth@staging.ottg.co.uk rm db.sqlite3
```

The `ssh` command takes an arbitrary shell command to run as its last argument, so we pass in `rm db.sqlite3`. We don't need a full path because we keep the SQLite database in our home folder.

Try not to accidentally delete your production database.

Wrap-Up: git tag the New Release

The last thing to do is to tag the release in our version control system (VCS)—it's important that we're always able to keep track of what's live:

```
$ git tag -f LIVE  # needs the -f because we are replacing the old tag
$ export TAG=`date +DEPLOYED-%F/%H%M`
$ git tag $TAG
$ git push -f origin LIVE $TAG
```

Some people don't like to use `push -f` and update an existing tag, and will instead use some kind of version number to tag their releases. Use whatever works for you.

And on that note, we can wrap up the last of the concepts we discussed in Part III, and move on to the more exciting topics that comprise Part IV. Can't wait!

Deployment Procedure Review

We've done a couple of deploys now, so this is a good time for a little recap:

- Deploy to staging first.
- Run our FTs against staging.
- Deploy to live.
- Tag the release.

Deployment procedures evolve and get more complex as projects grow, and it's an area that can become hard to maintain—full of manual checks and procedures—if you're not careful to keep things automated. There's lots more to learn about this, but it's out of scope for this book. Dave Farley's video on continuous delivery is a good place to start.

CHAPTER 19

User Authentication, Spiking, and De-Spiking

Our beautiful lists site has been live for a few days, and our users are starting to come back to us with feedback. "We love the site", they say, "but we keep losing our lists. Manually remembering URLs is hard. It'd be great if it could remember what lists we'd started."

Remember Henry Ford and faster horses. Whenever you hear a user requirement, it's important to dig a little deeper and think—what is the real requirement here? And how can I make it involve a cool new technology I've been wanting to try out?

Clearly the requirement here is that people want to have some kind of user account on the site. So, without further ado, let's dive into authentication.

Naturally we're not going to mess about with remembering passwords ourselves—besides being *so* '90s, secure storage of user passwords is a security nightmare we'd rather leave to someone else. We'll use something fun called "passwordless authentication" instead.[1]

Passwordless Auth with "Magic Links"

What authentication system could we use to avoid storing passwords ourselves? OAuth? OpenID? "Sign in with Facebook"? Ugh. For me, those all have unacceptable creepy overtones; why should Google or Facebook know what sites you're logging in to and when?

1 If you *insist* on storing your own passwords, Django's default authentication module is ready and waiting for you. It's nice and straightforward, and I'll leave it to you to discover on your own.

Instead, for the second edition,[2] I found a fun approach to authentication that now goes by the name of "Magic Links", but you might call it "just use email".

The system was invented (or at least popularised) back in 2014 by someone annoyed at having to create new passwords for so many websites. They found themselves just using random, throwaway passwords, not even trying to remember them, and using the "forgot my password" feature whenever they needed to log in again. You can read all about it on Medium.

The concept is: just use email to verify someone's identity. If you're going to have a "forgot my password" feature, then you're trusting email anyway, so why not just go the whole hog? Whenever someone wants to log in, we generate a unique URL for them to use, email it to them, and they then click through that to get into the site.

It's by no means a perfect system, and in fact there are lots of subtleties to be thought through before it would really make a good login solution for a production website, but this is just a fun toy project so let's give it a go.

A Somewhat Larger Spike

Reminder: a spike is a phase of exploratory coding, where we can code without tests, in order to explore a new tool or experiment with a new idea. We will come back and redo the code "properly" with TDD later.

To get this Magic Links project set up, the first thing I did was take a look at existing Python and Django authentication packages, like django-allauth, but both of them looked overcomplicated for this stage (and besides, it'll be more fun to code our own!).

So instead, I dived in and hacked about, and after a few dead ends and wrong turns, I had something that just about works. I'll take you on a tour, and then we'll go through and "de-spike" the implementation—that is, replace the prototype with tested, production-ready code.

You should go ahead and add this code to your own site too, and then you can have a play with it. Try logging in with your own email address, and convince yourself that it really does work.

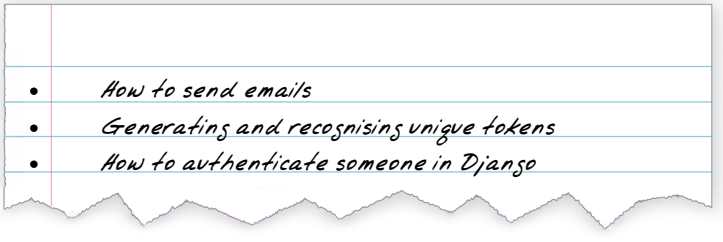
2 In the first edition, I used an experimental project called "Persona", cooked up by some of the wonderful techno-hippie-idealists at Mozilla, but sadly that project was abandoned.

Starting a Branch for the Spike

This spike is going to be a bit more involved than the last one, so we'll be a little more rigorous with our version control. Before embarking on a spike it's a good idea to start a new branch, so you can still use your VCS without worrying about your spike commits getting mixed up with your production code:

```
$ git switch -c passwordless-spike
```

Let's keep track of some of the things we're hoping to learn from the spike:



Frontend Login UI

Let's start with the frontend by adding in an actual form to enter your email address into the navbar, along with a logout link for users who are already authenticated:

src/lists/templates/base.html (ch19l001)

```
<body>
  <div class="container">

    <div class="navbar">
      {% if user.is_authenticated %}
        <p>Logged in as {{ user.email }}</p>
        <form method="POST" action="/accounts/logout">
          {% csrf_token %}
          <button id="id_logout" type="submit">Log out</button>
        </form>
      {% else %}
        <form method="POST" action ="accounts/send_login_email">
          Enter email to log in: <input name="email" type="text" />
          {% csrf_token %}
        </form>
      {% endif %}
    </div>

    <div class="row justify-content-center p-5 bg-body-tertiary rounded-3">
    [...]
```

Sending Emails from Django

The login will be something like Figure 19-1.

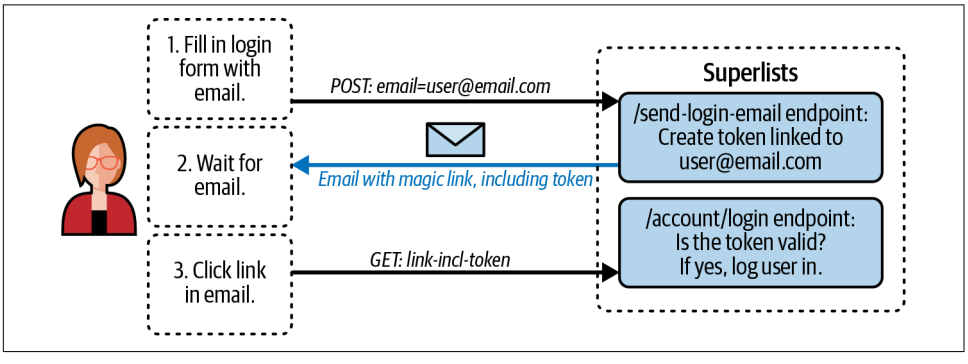


Figure 19-1. Overview of the Magic Links login process

1. When someone wants to log in, we generate a unique secret token for them, link it to their email, store it in the database, and send it to them.
2. The user then checks their email, which will have a link for a URL that includes that token.
3. When they click that link, we check whether the token exists in the database and, if so, they are logged in as the associated user.

First, let's prep an app for our accounts stuff:

```
$ cd src && python manage.py startapp accounts && cd ..
$ ls src/accounts
__init__.py admin.py    apps.py     migrations  models.py   tests.py    views.py
```

And we'll wire up *urls.py* with at least one URL. In the top-level *superlists/urls.py*…

src/superlists/urls.py (ch19l003)

```
from django.urls import include, path
from lists import views as list_views

urlpatterns = [
    path("", list_views.home_page, name="home"),
    path("lists/", include("lists.urls")),
    path("accounts/", include("accounts.urls")),
]
```

And we give the accounts module its own *urls.py*:

src/accounts/urls.py (ch19l004)

```
from django.urls import path

from accounts import views

urlpatterns = [
    path("send_login_email", views.send_login_email, name="send_login_email"),
]
```

Here's the view that's in charge of creating a token associated with the email address that the user puts in our login form:

src/accounts/views.py (ch19l005)

```
import sys
import uuid

from django.core.mail import send_mail
from django.shortcuts import render

from accounts.models import Token


def send_login_email(request):
    email = request.POST["email"]
    uid = str(uuid.uuid4())
    Token.objects.create(email=email, uid=uid)
    print("saving uid", uid, "for email", email, file=sys.stderr)
    url = request.build_absolute_uri(f"/accounts/login?uid={uid}")
    send_mail(
        "Your login link for Superlists",
        f"Use this link to log in:\n\n{url}",
        "noreply@superlists",
        [email],
    )
    return render(request, "login_email_sent.html")
```

For that to work, we'll need a template with a placeholder message confirming the email was sent:

src/accounts/templates/login_email_sent.html (ch19l006)

```
<html>
<h1>Email sent</h1>

<p>Check your email, you'll find a message with a link that will log you into
the site.</p>

</html>
```

(You can see how hacky this code is—we'd want to integrate this template with our *base.html* in the real version.)

Email Server Config for Django

The django docs on email explain how `send_mail()` works, as well as how you configure it by telling Django what email server to use, and how to authenticate with it. Here, I'm just using my Gmail[3] account for now—but you can use any email provider you like, as long as they support SMTP (Simple Mail Transfer Protocol):

src/superlists/settings.py (ch19l007)

```
EMAIL_HOST = "smtp.gmail.com"
EMAIL_HOST_USER = "obeythetestinggoat@gmail.com"
EMAIL_HOST_PASSWORD = os.environ.get("EMAIL_PASSWORD")
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

If you want to use Gmail as well, you'll probably have to visit your Google account security settings page. If you're using two-factor authentication, you'll want to set up an app-specific password. If you're not, you will probably still need to allow access for less secure apps. You might want to consider creating a new Google account for this purpose, rather than using one containing sensitive data.

Another Secret, Another Environment Variable

Once again, we have a "secret" that we want to avoid keeping directly in our source code or on GitHub, so another environment variable is used in the `os.environ.get`. To get this to work, we need to set it in the shell that's running my dev server:

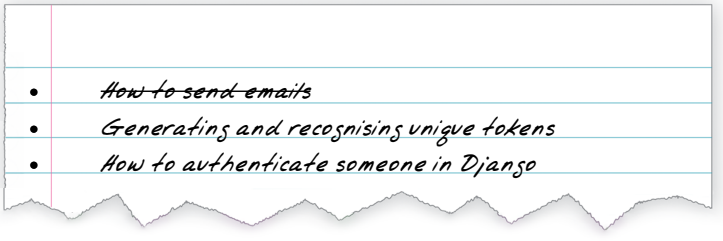
```
$ export EMAIL_PASSWORD="ur-email-server-password-here"
```

Later, we'll see about adding that to the env file on the staging server as well.

3 Didn't I just spend a whole intro banging on about the privacy implications of using Google for login, only to go on and use Gmail? Yes, it's a contradiction (honest, I will move off Gmail one day!). But in this case I'm just using it for testing, and the important thing is that I'm not forcing Google on my users.

Storing Tokens in the Database

How are we doing? Let's review where we're at in the process:



We'll need a model to store our tokens in the database—they link an email address with a unique ID. It's pretty simple:

src/accounts/models.py (ch19l008)

```
from django.db import models


class Token(models.Model):
    email = models.EmailField()
    uid = models.CharField(max_length=255) ❶
```

❶ Django does have a specific UID (universally unique identifier) fields type for many databases, but I just want to keep things simple for now.

The point of this spike is about authentication and emails, not optimising database storage. We've got enough things we need to learn as it is!

Let's switch on our new accounts app in *settings.py*:

src/superlists/settings.py (ch19l008-1)

```
INSTALLED_APPS = [
    # "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "lists",
    "accounts",
]
```

We can then do a quick migrations dance to add the token model to the database:

```
$ python src/manage.py makemigrations
Migrations for 'accounts':
  src/accounts/migrations/0001_initial.py
    + Create model Token
$ python src/manage.py migrate
Operations to perform:
  Apply all migrations: accounts, auth, contenttypes, lists, sessions
Running migrations:
  Applying accounts.0001_initial... OK
```

And at this point, if you actually try the email form in your browser, you'll see it really does send an actual real email—to your real email address hopefully (best not spam someone else now!). See Figures 19-2 and 19-3.

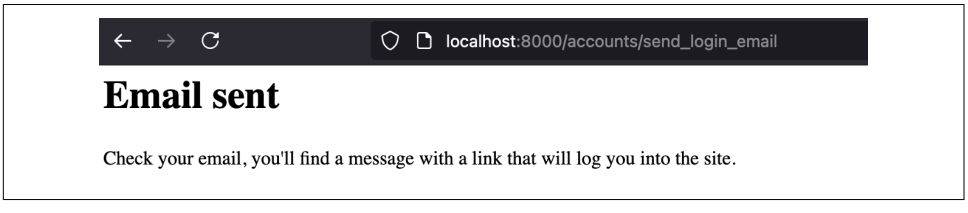


Figure 19-2. Looks like we might have sent an email

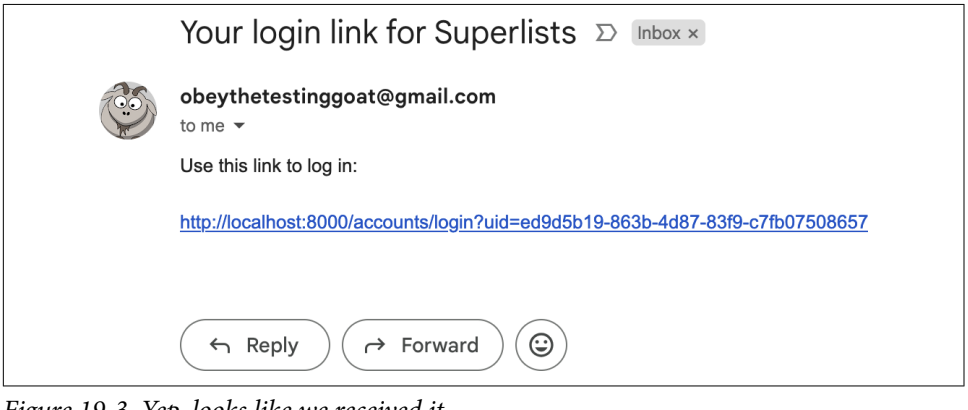
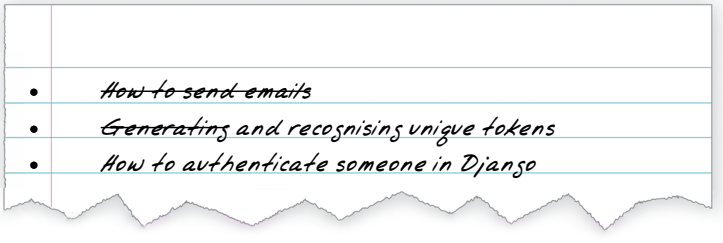


Figure 19-3. Yep, looks like we received it

Custom Authentication Models

OK, so we've done the first half of "Generating and recognising unique tokens":



But, before we can move on to recognising them and making the login work end-to-end though, we need to explore Django's authentication system. The first thing we'll need is a user model. I took a dive into the Django auth documentation and tried to hack in the simplest possible one:

src/accounts/models.py (ch19l009)

```
from django.contrib.auth.models import (
    AbstractBaseUser,
    BaseUserManager,
)
[...]


class ListUser(AbstractBaseUser):
    email = models.EmailField(primary_key=True)
    USERNAME_FIELD = "email"
    # REQUIRED_FIELDS = ['email', 'height']

    objects = ListUserManager()

    @property
    def is_staff(self):
        return self.email == "harry.percival@example.com"

    @property
    def is_active(self):
        return True
```

That's what I call a minimal user model! One field, none of this first name/last name/username nonsense, and—pointedly—no password! That's somebody else's problem!

But, again, you can see that this code isn't ready for production—from the commented-out lines to the hardcoded Harry email address. We'll neaten this up quite a lot when we de-spike.

To get it to work, I needed to add a model manager for the user, for some reason:

src/accounts/models.py (ch19l010)

```
[...]
class ListUserManager(BaseUserManager):
    def create_user(self, email):
        ListUser.objects.create(email=email)

    def create_superuser(self, email, password):
        self.create_user(email)
```

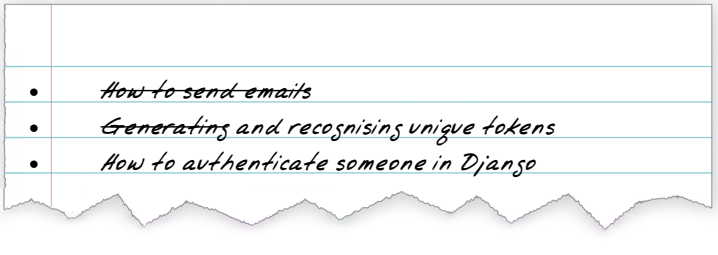
No need to worry about what a model manager is at this stage; for now, we just need it because we need it, and it works. When we de-spike, we'll examine each bit of code that actually ends up in production and make sure we understand it fully.

We'll need to run `makemigrations` and `migrate` again to make the user model real:

```
$ python src/manage.py makemigrations
Migrations for 'accounts':
  src/accounts/migrations/0002_listuser.py
    + Create model ListUser
$ python src/manage.py migrate
[...]
Running migrations:
  Applying accounts.0002_listuser... OK
```

Finishing the Custom Django Auth

Let's review our scratchpad:



Hmm, we can't quite cross off anything yet. Turns out the steps we *thought* we'd go through aren't quite the same as the steps we're *actually* going through (this is not uncommon, as I'm sure you know).

Still, we're almost there—our last step will combine recognising the token and then actually logging the user in. Once we've done this, we'll be able to pretty much strike off all the items on our scratchpad.

So here's the view that actually handles the click-through from the link in the email:

src/accounts/views.py (ch19l011)

```
import sys
import uuid

from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.core.mail import send_mail
from django.shortcuts import redirect, render

from accounts.models import Token


def send_login_email(request):
    [...]


def login(request):
    print("login view", file=sys.stderr)
    uid = request.GET.get("uid")
    user = authenticate(request, uid=uid)
    if user is not None:
        auth_login(request, user)
    return redirect("/")
```

The `authenticate()` function invokes Django's authentication framework, which we configure using a custom "authentication backend," whose job it is to validate the UID (unique identifier) and return a user with the right email.

We could have done this stuff directly in the view, but we may as well structure things the way Django expects. It makes for a reasonably neat separation of concerns:

src/accounts/authentication.py (ch19l012)

```
import sys

from accounts.models import ListUser, Token

from django.contrib.auth.backends import BaseBackend


class PasswordlessAuthenticationBackend(BaseBackend):
    def authenticate(self, request, uid):
        print("uid", uid, file=sys.stderr)
        if not Token.objects.filter(uid=uid).exists():
            print("no token found", file=sys.stderr)
            return None
        token = Token.objects.get(uid=uid)
        print("got token", file=sys.stderr)
        try:
            user = ListUser.objects.get(email=token.email)
            print("got user", file=sys.stderr)
            return user
        except ListUser.DoesNotExist:
            print("new user", file=sys.stderr)
            return ListUser.objects.create(email=token.email)

    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

Again, lots of debug prints in there, and some duplicated code—not something we'd want in production, but it works…as long as we add it to *settings.py* (it doesn't matter where):

src/superlists/settings.py (ch19l012-1)

```
AUTH_USER_MODEL = "accounts.ListUser"
AUTHENTICATION_BACKENDS = [
    "accounts.authentication.PasswordlessAuthenticationBackend",
]
```

And finally, a logout view:

src/accounts/views.py (ch19l013)

```
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.contrib.auth import logout as auth_logout
[...]


def logout(request):
    auth_logout(request)
    return redirect("/")
```

Add login and logout to our *urls.py*…

src/accounts/urls.py (ch19l014)

```
urlpatterns = [
    path("send_login_email", views.send_login_email, name="send_login_email"),
    path("login", views.login, name="login"),
    path("logout", views.logout, name="logout"),
]
```

And we should be all done! Spin up a dev server with `runserver` and try it—believe it or not, it *actually* works (see Figure 19-4).

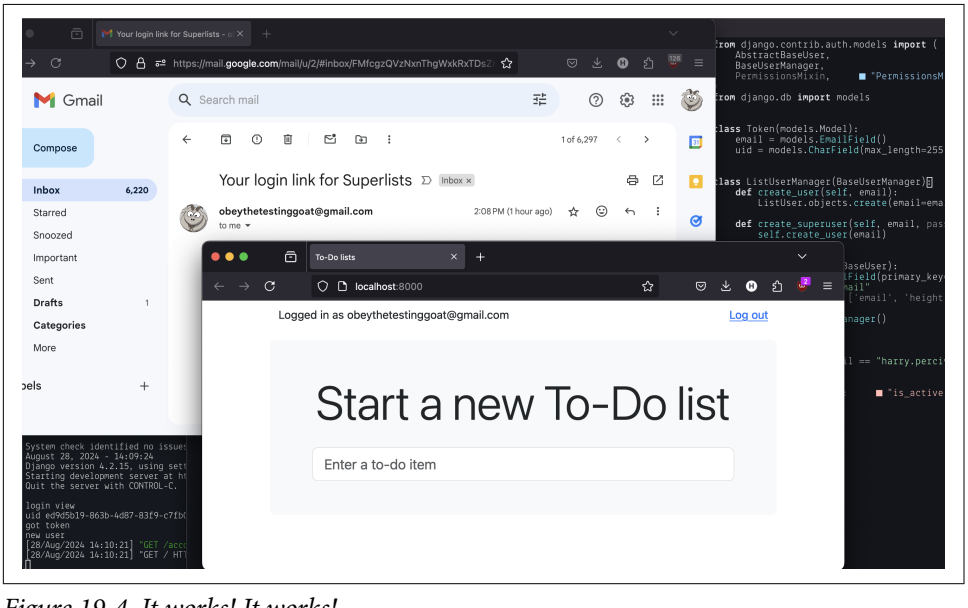


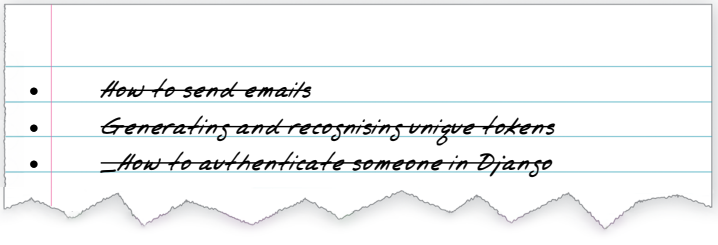
Figure 19-4. It works! It works!

If you get an `SMTPSenderRefused` error message, don't forget to set the `EMAIL_PASSWORD` environment variable in the shell that's running `runserver`. Also, if you see a message saying "Application-specific password required", that's a Gmail security policy. Follow the link in the error message.

That's pretty much it! Along the way, I had to fight pretty hard, including clicking around the Gmail account security UI for a while, stumbling over several missing attributes on my custom user model (because I didn't read the docs properly), and even at one point switching to the dev version of Django to overcome a bug, which thankfully turned out to be a red herring.

But we now have a working solution! Let's commit it on our spike branch:

```
$ git status
$ git add src/accounts
$ git commit -am "spiked in custom passwordless auth backend"
```



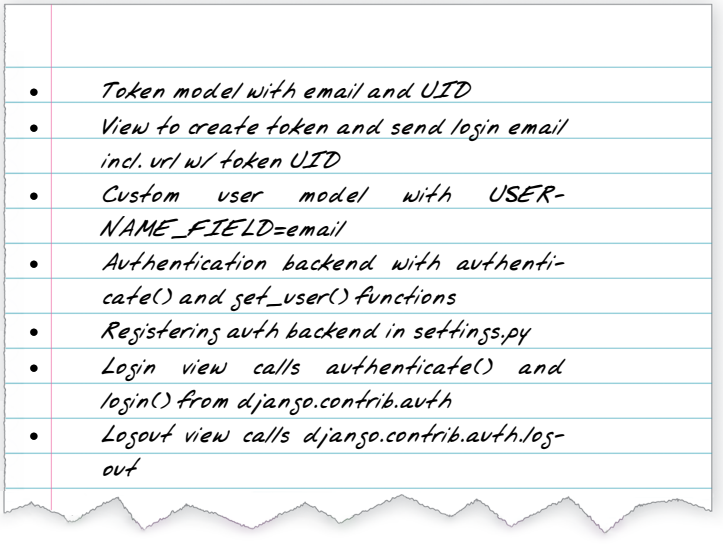
Time to de-spike!

De-Spiking

De-spiking means rewriting your prototype code using TDD. In this section, we'll work through how to do that in a safe and methodical way. We'll take the knowledge we've acquired during the spiking process—whether that's in our heads, in our notes, or in our branch in Git—and apply it as we re-implement gradually in a test-first way. And the hope is that our code will turn out a bit nicer the second time around!

Making a Plan

While it's fresh in our minds, let's make a few notes based on what we've learned about what we know we're probably going to need to build during our de-spike:



Wring an FT Against the Spiked Code

We now have enough information to "do it properly". So, what's the first step? An FT, of course! We'll stay on the spike branch for now to see our FT pass against our spiked code. Then we'll go back to our main branch and commit just the FT.

Here's a first, simple version of the FT:

src/functional_tests/test_login.py (ch19l018)

```
import re

from django.core import mail
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

from .base import FunctionalTest

TEST_EMAIL = "edith@example.com"  ❶
SUBJECT = "Your login link for Superlists"


class LoginTest(FunctionalTest):
    def test_login_using_magic_link(self):
        # Edith goes to the awesome superlists site
        # and notices a "Log in" section in the navbar for the first time
        # It's telling her to enter her email address, so she does
        self.browser.get(self.live_server_url)
        self.browser.find_element(By.CSS_SELECTOR, "input[name=email]").send_keys(
            TEST_EMAIL, Keys.ENTER
        )

        # A message appears telling her an email has been sent
        self.wait_for(
            lambda: self.assertIn(
                "Check your email",
                self.browser.find_element(By.CSS_SELECTOR, "body").text,
            )
        )

        # She checks her email and finds a message
        email = mail.outbox.pop()  ❷
        self.assertIn(TEST_EMAIL, email.to)
        self.assertEqual(email.subject, SUBJECT)

        # It has a URL link in it
        self.assertIn("Use this link to log in", email.body)
        url_search = re.search(r"http://.+/.+$", email.body)
        if not url_search:
            self.fail(f"Could not find url in email body:\n{email.body}")
        url = url_search.group(0)
        self.assertIn(self.live_server_url, url)

        # she clicks it
        self.browser.get(url)

        # she is logged in!
        self.wait_for(
            lambda: self.browser.find_element(By.CSS_SELECTOR, "#id_logout"),
        )
        navbar = self.browser.find_element(By.CSS_SELECTOR, ".navbar")
        self.assertIn(TEST_EMAIL, navbar.text)
```

❶ Whenever you're testing against something that can send real emails, you don't want to use a real address. It's best practice to use a special domain like `@example.com`, which has been reserved for exactly this sort of thing, to avoid accidentally spamming anyone!

❷ Were you worried about how we were going to handle retrieving emails in our tests? Thankfully, we can cheat for now! When running tests, Django gives us access to any emails that the server tries to send via the `mail.outbox` attribute. We'll discuss checking "real" emails in Chapter 23.

And if we run the FT, it works!

```
$ python src/manage.py test functional_tests.test_login
[...]
Not Found: /favicon.ico
saving uid [...]
login view
uid [...]
got token
new user

.
 ---------------------------------------------------------------------
Ran 1 test in 2.729s

OK
```

You can even see some of the debug output I left in my spiked view implementations. Now it's time to revert all of our temporary changes, and reintroduce them one by one in a test-driven way.

Reverting Our Spiked Code

We can revert our spike using our version control system:

```
$ git switch main # switch back to main branch
$ rm -rf src/accounts # remove any trace of spiked code
$ git add src/functional_tests/test_login.py
$ git commit -m "FT for login via email"
```

Now we rerun the FT and let it be the main driver of our development, referring back to our scratchpad from time to time when we need to:

```
$ python src/manage.py test functional_tests.test_login
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: input[name=email]; [...]
[...]
```

If you see an exception saying "No module named accounts", you may have missed a step in the de-spiking process—maybe a commit or the change of branch.

The first thing it wants us to do is add an email input element. Bootstrap has some built-in classes for navigation bars, so we'll use them, and include a form for the login email:[4]

src/lists/templates/base.html (ch19l020)

```
<body>
  <div class="container">

    <nav class="navbar">
      <div class="container-fluid">
        <a class="navbar-brand" href="/">Superlists</a>
        <form method="POST" action="/accounts/send_login_email">
          <div class="input-group">
            <label class="navbar-text me-2" for="id_email_input">
              Enter your email to log in
            </label>
            <input
              id="id_email_input"
              name="email"
              class="form-control"
              placeholder="your@email.com"
            />
            {% csrf_token %}
          </div>
        </form>
      </div>
    </nav>


    <div class="row justify-content-center p-5 bg-body-tertiary rounded-3">
      <div class="col-lg-6 text-center">
        <h1 class="display-1 mb-4">{% block header_text %}{% endblock %}</h1>
        [...]
```

4 We are now introducing a conceptual dependency from the base template to the `accounts` app because its URL is in the form. I didn't want to spend time on this in the book, but this might be a good time to consider moving the template out of *lists/templates* and into *superlists/templates*. By convention, that's the place for templates whose scope is wider than a single app.

At this point, you'll find that the unit tests start to fail:

```
ERROR: test_renders_input_form
[...]
    [form] = parsed.cssselect("form[method=POST]")
    ^^^^^^
ValueError: too many values to unpack (expected 1, got 2)


ERROR: test_renders_input_form
    [form] = parsed.cssselect("form[method=POST]")
    ^^^^^^
ValueError: too many values to unpack (expected 1, got 2)
```

It's because these unit tests had a hard assumption that there's only one POST form on the page. Let's change them to be more resilient. Here's how you might change the first one:

src/lists/tests/test_views.py (ch19l020-1)

```
    def test_renders_input_form(self):
        response = self.client.get("/")
        parsed = lxml.html.fromstring(response.content)
        forms = parsed.cssselect("form[method=POST]")  ❶
        self.assertIn("/lists/new", [form.get("action") for form in forms])  ❷
        [form] = [form for form in forms if form.get("action") == "/lists/new"]  ❸
        inputs = form.cssselect("input")  ❹
        self.assertIn("text", [input.get("name") for input in inputs])  ❹
```

❶ We get all forms, rather than using the clever `[form] =` syntax.

❷ We check that at least *one* of the forms has the right `action=` URL. I'm using `assertIn()`, so we get a nice error message. If we can't find the right URL, we'll see the list of URLs that *do* exist on the page.

❸ Now we can feel free to go back to unpacking, and get the right form, based on its `action` attribute.

❹ The rest of the test is as before.

Here's a similar set of changes in the second test:

src/lists/tests/test_views.py (ch19l020-2)

```
@@ -65,10 +65,12 @@ class ListViewTest(TestCase):

     def test_renders_input_form(self):
         mylist = List.objects.create()
-        response = self.client.get(f"/lists/{mylist.id}/")
+        url = f"/lists/{mylist.id}/"
+        response = self.client.get(url)
         parsed = lxml.html.fromstring(response.content)
-        [form] = parsed.cssselect("form[method=POST]")
-        self.assertEqual(form.get("action"), f"/lists/{mylist.id}/")
+        forms = parsed.cssselect("form[method=POST]")
+        self.assertIn(url, [form.get("action") for form in forms])
+        [form] = [form for form in forms if form.get("action") == url]
         inputs = form.cssselect("input")
         self.assertIn("text", [input.get("name") for input in inputs])
```

It's pretty much the same edit, except this time I decided to have a `url` variable, to remove the duplication of using `/lists/{mylist.id}/` three times. That gets our unit tests passing again:

```
OK
```

If we try our FT again, we'll see it fails because the login form doesn't send us to a real URL yet—you'll see the `Not found:` message in the server output, as well as the assertion reporting the content of the default 404 page:

```
$ python src/manage.py test functional_tests.test_login
[...]
Not Found: /accounts/send_login_email
[...]
AssertionError: 'Check your email' not found in 'Not Found\nThe requested
resource was not found on this server.'
```

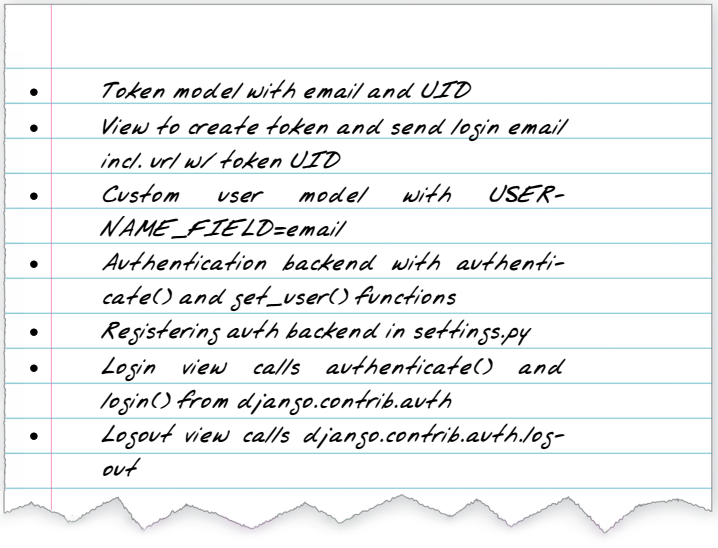
Time to start writing some Django code. We begin, like in the spike, by creating an app called `accounts` to hold all the files related to login:

```
$ cd src && python manage.py startapp accounts && cd ..
$ ls src/accounts
__init__.py admin.py    apps.py     migrations  models.py   tests.py    views.py
```

You could even do a commit just for that, to be able to distinguish the placeholder app files from our modifications.

A Minimal Custom User Model

Let's turn to the models layer:[5]



We know we have to build a token model and a custom user model, and the user model was the messiest part in our spike. So, let's have a go at redoing that test-first, to see if it comes out nicer.

Django's built-in user model makes all sorts of assumptions about what information you want to track about users—from explicitly requiring a first name and last name[6] to forcing you to use a username. I'm a great believer in not storing information about users unless you absolutely must, so a user model that records an email address and nothing else sounds good to me!

Let's start straight away with a tests folder instead of *tests.py* in this app:

```
$ rm src/accounts/tests.py
$ mkdir src/accounts/tests
$ touch src/accounts/tests/__init__.py
```

5 In this chapter, we're building things in a "bottom-up" way, starting with the models, and then building the layers on top—the views and templates that depend on them. This is a common approach, but it's not the only one! In Chapter 24 we'll explore building software from the outside in, which has all sorts of advantages too.

6 This is a decision that even some prominent Django maintainers have said they now regret—not everyone has a first and last name.

And now, let's add a *test_models.py* to say:

src/accounts/tests/test_models.py (ch19l023)

```
from django.test import TestCase

from accounts.models import User


class UserModelTest(TestCase):
    def test_user_is_valid_with_email_only(self):
        user = User(email="a@b.com")
        user.full_clean()  # should not raise
```

That gives us the expected failure:

```
$ python src/manage.py test accounts
[...]
ImportError: cannot import name 'User' from 'accounts.models'
(...goat-book/src/accounts/models.py)
```

OK, let's try the absolute minimum then:

src/accounts/models.py (ch19l024)

```
from django.db import models


class User(models.Model):
    email = models.EmailField()
```

That gives us an error because Django won't recognise models unless they're in `INSTALLED_APPS`:

```
RuntimeError: Model class accounts.models.User doesn't declare an explicit
app_label and isn't in an application in INSTALLED_APPS.
```

So, let's add it to *settings.py*:

src/superlists/settings.py (ch19l025)

```
INSTALLED_APPS = [
    # "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "accounts",
    "lists",
]
```

And that gets our tests passing!

```
OK
```

Now let's see if we've built a user model that Django can actually work with. There's a built-in function in `django.contrib.auth` called `get_user_model()`—which retrieves the currently active user model and, as we'll see, also performs some checks on it. Let's use it in our tests:

src/accounts/tests/test_models.py (ch19l026-1)

```
from django.contrib import auth
from django.test import TestCase

from accounts.models import User


class UserModelTest(TestCase):
    def test_model_is_configured_for_django_auth(self):
        self.assertEqual(auth.get_user_model(), User)

    def test_user_is_valid_with_email_only(self):
        [...]
```

That gives:

```
AssertionError: <class 'django.contrib.auth.models.User'> != <class
'accounts.models.User'>
```

OK, so let's try wiring up our model inside *settings.py*, in a variable called `AUTH_USER_MODEL`:

src/superlists/settings.py (ch19l026-2)

```
AUTH_USER_MODEL = "accounts.User"
```

Now when we run our tests, Django complains that our custom user model is missing a couple of bits of metadata. In fact, it's so unhappy that it won't even run the tests:

```
$ python src/manage.py test accounts
Traceback (most recent call last):
[...]
  File ".../django/contrib/auth/checks.py", line 46, in check_user_model
    if not isinstance(cls.REQUIRED_FIELDS, (list, tuple)):
                      ^^^^^^^^^^^^^^^^^^^
AttributeError: type object 'User' has no attribute 'REQUIRED_FIELDS'
```

Sigh. Come on, Django; it's only got one field, so you should be able to figure out the answers to these questions for yourself.

Here you go:

src/accounts/models.py (ch19l027)

```
class User(models.Model):
    email = models.EmailField()

    REQUIRED_FIELDS = []
```

Next silly question?[7]

```
AttributeError: type object 'User' has no attribute 'USERNAME_FIELD'
```

We'll go through a few more of these, until we get to:

src/accounts/models.py (ch19l029)

```
class User(models.Model):
    email = models.EmailField()

    REQUIRED_FIELDS = []
    USERNAME_FIELD = "email"
    is_anonymous = False
    is_authenticated = True
```

And now we get a slightly different error:

```
$ python src/manage.py test accounts
[...]
SystemCheckError: System check identified some issues:

ERRORS:
accounts.User: (auth.E003) 'User.email' must be unique because it is named as
the 'USERNAME_FIELD'.
```

Well, the simple way to fix that would be like this:

src/accounts/models.py (ch19l030)

```
    email = models.EmailField(unique=True)
```

And now we get a different error again, slightly more familiar this time! Django is a bit happier with the structure of our custom user model, but it's unhappy about the database:

```
django.db.utils.OperationalError: no such table: accounts_user
```

7 You might ask, if I think Django is so silly, why don't I submit a pull request to fix it? It should be quite a simple fix. Well, I promise I will, as soon as I've finished updating the book. For now, snarky comments will have to suffice.

In other words, we need to create a migration:

```
$ python src/manage.py makemigrations
Migrations for 'accounts':
  src/accounts/migrations/0001_initial.py
    + Create model User
```

And our tests pass:

```
$ python src/manage.py test accounts
[...]
Ran 2 tests in 0.001s
OK
```

But our model isn't quite as simple as it could be. It has the email field, and also an autogenerated "ID" field as its primary key. We could make it even simpler!

Tests as Documentation

Let's go all the way and make the email field the primary key,[8] and thus implicitly remove the autogenerated `id` column. Although we could just *do it* and our test would still pass, and conceivably claim it was "just a refactor", it would be better to have a specific test:

src/accounts/tests/test_models.py (ch19l032)

```
class UserModelTest(TestCase):
    def test_model_is_configured_for_django_auth(self):
        [...]
    def test_user_is_valid_with_email_only(self):
        [...]

    def test_email_is_primary_key(self):
        user = User(email="a@b.com")
        self.assertEqual(user.pk, "a@b.com")
```

It'll help us remember if we ever come back and look at the code again in future:

```
        self.assertEqual(user.pk, "a@b.com")
AssertionError: None != 'a@b.com'
```

8 Emails may not be the perfect primary key in real life. One reader—clearly deeply scarred—wrote me an emotional email about how much they've suffered for over a decade from trying to deal with the consquences of using email as a primary key, particularly how it makes multiuser account management nearly impossible. So, as ever, YMMV.

Your tests can be a form of documentation for your code—they express the requirements for a particular class or function. Sometimes, if you forget why you've done something a particular way, going back and looking at the tests will give you the answer. That's why it's important to make your tests readable, including giving them explicit, verbose method names.

Here's the implementation (`primary_key` makes the `unique=True` obsolete):

src/accounts/models.py (ch19l033)

```
    email = models.EmailField(primary_key=True)
```

And we mustn't forget to adjust our migrations:

```
$ rm src/accounts/migrations/0001_initial.py
$ python src/manage.py makemigrations
Migrations for 'accounts':
  src/accounts/migrations/0001_initial.py
    + Create model User
```

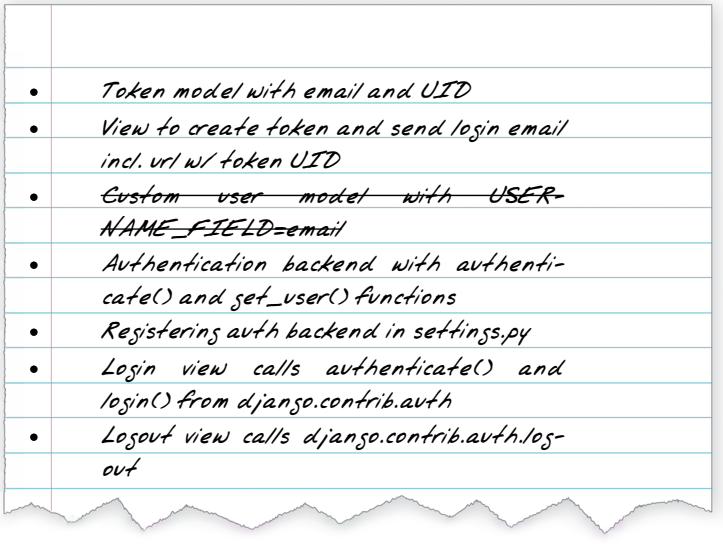
Now both our tests pass:

```
$ python src/manage.py test accounts
[...]
Ran 3 tests in 0.001s
OK
```

It's probably a good time for a commit, too:

```
$ git add src/accounts
$ git commit -m "custom user model with email as primary key"
```

And we can cross off one item from our de-spiking list. Hooray!



A Token Model to Link Emails with a Unique ID

Next let's build a token model. Here's a short unit test that captures the essence—you should be able to link an email to a unique ID, and that ID shouldn't be the same twice in a row:

src/accounts/tests/test_models.py (ch19l035)

```
from accounts.models import Token, User
[...]


class TokenModelTest(TestCase):
    def test_links_user_with_auto_generated_uid(self):
        token1 = Token.objects.create(email="a@b.com")
        token2 = Token.objects.create(email="a@b.com")
        self.assertNotEqual(token1.uid, token2.uid)
```

I won't show every single listing for creating the token class in *models.py*; I'll let you do that yourself instead. Driving Django models with basic TDD involves jumping through a few hoops because of the migration, so you'll see a few iterations like this—minimal code change, make migrations, get new error, delete migrations, re-create new migrations, another code change, and so on…

```
$ python src/manage.py test accounts
[...]
TypeError: Token() got unexpected keyword arguments: 'email'
```

I'll trust you to go through these conscientiously—remember, I may not be able to see you, but the Testing Goat can!

You might go through a hoop like this one, for example, where you find yourself needing to create and then delete a migration for an incomplete solution:

```
$ python src/manage.py makemigrations
Migrations for 'accounts':
  src/accounts/migrations/0002_token.py
    + Create model Token
$ python src/manage.py test accounts
AttributeError: 'Token' object has no attribute 'uid'. Did you mean: 'id'?
$ rm src/accounts/migrations/0002_token.py
```

Eventually, you should get to this code…

src/accounts/models.py (ch19l038)

```
class Token(models.Model):
    email = models.EmailField()
    uid = models.CharField(max_length=40)
```

And this error:

```
$ python src/manage.py test accounts
[...]

    self.assertNotEqual(token1.uid, token2.uid)
AssertionError: '' == ''
```

And here we have to decide how to generate our random unique ID field. We could use the `random` module, but Python actually comes with another module specifically designed for generating unique IDs called "UUID" (for "universally unique ID"). We can use it like this:

src/accounts/models.py (ch19l040)

```
import uuid
[...]

class Token(models.Model):
    email = models.EmailField()
    uid = models.CharField(default=uuid.uuid4, max_length=40)  ❶
```

❶ The `default=` argument for a field can be either a static value or a callable that returns a value at the time the model is created. In our case, using a callable means we'll get a different unique ID for every model.

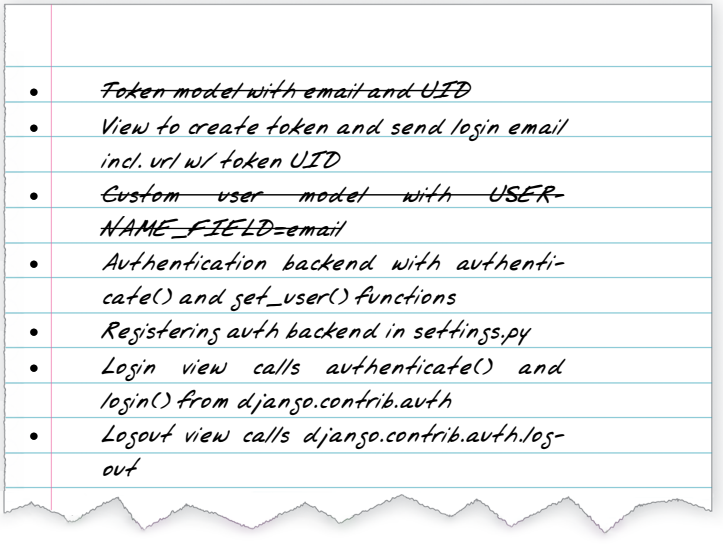
And, perhaps with a bit more wrangling of `makemigrations`…

```
$ rm src/accounts/migrations/0002_token.py
$ python src/manage.py makemigrations
Migrations for 'accounts':
  src/accounts/migrations/0002_token.py
    + Create model Token
```

…that should get us to passing tests:

```
$ python src/manage.py test accounts
[...]
Ran 4 tests in 0.015s

OK
```

So, we are well on our way!



The models layer is done, at least. In the next chapter, we'll get into mocking—a key technique for testing external dependencies like email.

Exploratory Coding, Spiking, and De-Spiking

Spiking
: Spiking is exploratory coding to find out about a new API, or to explore the feasibility of a new solution. Spiking can be done without tests. It's a good idea to do your spike on a new branch, and go back to your main branch when de-spiking.

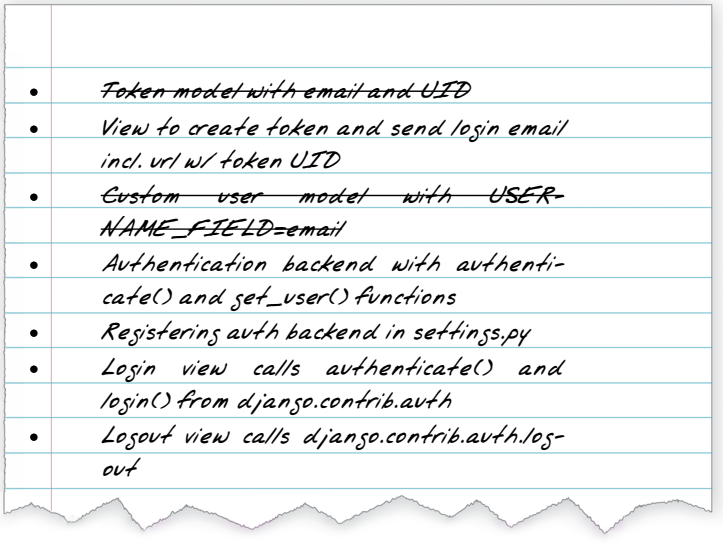
De-spiking
: De-spiking means taking the work from a spike and making it part of the production codebase. The idea is to throw away the old spike code altogether, and start again from scratch, using TDD once again. De-spiked code can often come out looking quite different from the original spike, and usually much nicer.

Writing your FT against spiked code
: Whether or not this is a good idea depends on your circumstances. The reason it can be useful is because it can help you write the FT correctly—figuring out how to test your spike can be just as challenging as the spike itself. On the other hand, it might constrain you to reimplementing a solution very similar to your spiked one; something to watch out for.

CHAPTER 20

Using Mocks to Test External Dependencies

In this chapter, we'll start testing the parts of our code that send emails—i.e., the second item on our scratchpad:



In the functional test (FT), you saw that Django gives us a way of retrieving any emails it sends by using the `mail.outbox` attribute. But in this chapter, I want to demonstrate a widespread testing technique called *mocking*. So, for the purpose of these unit tests, we'll pretend that this nice Django shortcut doesn't exist.

Am I telling you *not* to use Django's `mail.outbox`? No—use it; it's a neat helper. But I want to teach you about mocks because they're a useful general-purpose tool for unit testing external dependencies. You may not always be using Django! And even if you are, you may not be sending email—any interaction with a third-party API is a place you might find yourself wanting to test with mocks.

To Mock or Not to Mock?

I once gave a talk called "Stop Using Mocks"; it's entirely possible to find ways to write tests for external dependencies without using mocks at all.

I'm covering mocking in this book because it's such a common technique, but it does come with some downsides, as we'll see. Other techniques—including dependency injection and the use of custom fake objects—are well worth exploring, but they're more advanced.

My second book *Architecture Patterns with Python* goes into some detail on these alternatives.

Before We Start: Getting the Basic Plumbing In

Let's just get a basic view and URL set up first. We can do so with a simple test to ensure that our new URL for sending the login email should eventually redirect back to the home page:

src/accounts/tests/test_views.py (ch20l001)

```
from django.test import TestCase


class SendLoginEmailViewTest(TestCase):
    def test_redirects_to_home_page(self):
        response = self.client.post(
            "/accounts/send_login_email", data={"email": "edith@example.com"}
        )
        self.assertRedirects(response, "/")
```

Wire up the `include` in *superlists/urls.py*, plus the `url` in *accounts/urls.py*, and get the test passing with something a bit like this:

src/accounts/views.py (ch20l004)

```
from django.core.mail import send_mail  ❶
from django.shortcuts import redirect


def send_login_email(request):
    return redirect("/")
```

❶ I've added the import of the `send_mail` function as a placeholder for now.

If you've got the plumbing right, the tests should pass at this point:

```
$ python src/manage.py test accounts
[...]
Ran 5 tests in 0.015s

OK
```

OK, now we have a starting point—so let's get mocking!

Mocking Manually—aka Monkeypatching

When we call `send_mail` in real life, we expect Django to be making a connection to our email provider, and sending an actual email across the public internet. That's not something we want to happen in our tests. It's a similar problem whenever you have code that has external side effects—calling an API, sending out an SMS, integrating with a payment provider, whatever it may be.

When running our unit tests, we don't want to be sending out real payments or making API calls across the internet. But we would still like a way of testing that our code is correct. Mocks[1] give us one way to do that.

Actually, one of the great things about Python is that its dynamic nature makes it very easy to do things like mocking—or what's sometimes called monkeypatching. Let's suppose that, as a first step, we want to get to some code that invokes `send_mail` with the right subject line, "from" address, and "to" address. That would look something like this:

1 I'm using the generic term "mock", but testing enthusiasts like to distinguish other types of a general class of test tools called "test doubles", including spies, fakes, and stubs. The differences don't really matter for this book, but if you want to get into the nitty-gritty, check out the amazing wiki by Justin Searls. Warning: absolutely chock full of great testing content.

src/accounts/views.py (expected future contents)

```
def send_login_email(request):
    email = request.POST["email"]
    # expected future code:
    send_mail(
        "Your login link for Superlists",
        "some kind of body text tbc",
        "noreply@superlists",
        [email],
    )
    return redirect("/")
```

How can we test this without calling the *real* `send_mail` function? The answer is that our test can ask Python to swap out the `send_mail` function for a fake version, at runtime, just before we invoke the `send_login_email` view.

Check this out:

src/accounts/tests/test_views.py (ch20l005)

```
from django.test import TestCase

import accounts.views  ❷


class SendLoginEmailViewTest(TestCase):
    def test_redirects_to_home_page(self):
        [...]

    def test_sends_mail_to_address_from_post(self):
        self.send_mail_called = False

        def fake_send_mail(subject, body, from_email, to_list):  ❶
            self.send_mail_called = True
            self.subject = subject
            self.body = body
            self.from_email = from_email
            self.to_list = to_list

        accounts.views.send_mail = fake_send_mail  ❷

        self.client.post(
            "/accounts/send_login_email", data={"email": "edith@example.com"}
        )

        self.assertTrue(self.send_mail_called)
        self.assertEqual(self.subject, "Your login link for Superlists")
        self.assertEqual(self.from_email, "noreply@superlists")
        self.assertEqual(self.to_list, ["edith@example.com"])
```

❶ We define a `fake_send_mail` function, which looks like the real `send_mail` function, but all it does is save some information about how it was called, using some variables on `self`.

❷ Then, before we execute the code under test by doing the `self.client.post`, we swap out the real `accounts.views.send_mail` with our fake version—it's as simple as just assigning it.

It's important to realise that there isn't really anything magical going on here; we're just taking advantage of Python's dynamic nature and scoping rules.

Up until we actually invoke a function, we can modify the variables it has access to, as long as we get into the right namespace. That's why we import the top-level `accounts` module: to be able to get down to the `accounts.views` module, which is the scope in which the `accounts.views.send_login_email` function will run.

This isn't even something that only works inside unit tests—you can do this kind of monkeypatching in any Python code! That may take a little time to sink in. See if you can convince yourself that it's not all totally crazy—and then consider a couple of extra details that are worth knowing:

- Why do we use `self` as a way of passing information around? It's just a convenient variable that's available both inside the scope of the `fake_send_mail` function and outside of it. We could use any mutable object, like a list or a dictionary, as long as we are making in-place changes to an existing variable that exists outside our fake function. (Feel free to have a play around with different ways of doing this, if you're curious, and see what works and doesn't.)
- The "before" is critical! I can't tell you how many times I've sat there, wondering why a mock isn't working, only to realise that I didn't mock *before* I called the code under test.

Let's see if our hand-rolled mock object will let us test-drive some code:

```
$ python src/manage.py test accounts
[...]
    self.assertTrue(self.send_mail_called)
AssertionError: False is not true
```

So let's call `send_mail`, naively:

src/accounts/views.py (ch20l006-1)

```
from django.core.mail import send_mail  ❶
[...]

def send_login_email(request):
    send_mail()  ❷
    return redirect("/")
```

❶ This import should still be in the file from earlier, but in case an overenthusiastic IDE has removed it, I'm re-listing it for you here.

❷ Here's our new call to `send_mail()`.

That gives:

```
TypeError: SendLoginEmailViewTest.test_sends_mail_to_address_from_post.<locals>
.fake_send_mail() missing 4 required positional arguments: 'subject', 'body',
'from_email', and 'to_list'
```

It looks like our monkeypatch is working! We've called `send_mail`, and it's gone into our `fake_send_mail` function, which wants more arguments. Let's try this:

src/accounts/views.py (ch20l006-2)

```
def send_login_email(request):
    send_mail("subject", "body", "from_email", ["to email"])
    return redirect("/")
```

That gives:

```
    self.assertEqual(self.subject, "Your login link for Superlists")
AssertionError: 'subject' != 'Your login link for Superlists'
```

That's working pretty well! Now we can work step-by-step, all the way through to something like this:

src/accounts/views.py (ch20l006)

```
def send_login_email(request):
    email = request.POST["email"]
    send_mail(
        "Your login link for Superlists",
        "body text tbc",
        "noreply@superlists",
        [email],
    )
    return redirect("/")
```

And we have passing tests!

```
$ python src/manage.py test accounts

Ran 6 tests in 0.016s

OK
```

Brilliant! We've managed to write tests for some code, which would ordinarily go out and try to send real emails across the internet, and by "mocking out" the `send_email` function, we're able to write the tests and code all the same.[2]

But our hand-rolled mock has a couple of problems:

- It involved a fair bit of boilerplate code, populating all those `self.xyz` variables to let us assert on them.
- More importantly, although we didn't see this, the monkeypatching will persist from one test to the next, breaking isolation between tests. This can cause serious confusion.

The Python Mock Library

The `mock` package was added to the standard library as part of Python 3.3. It provides a magical object called a `Mock`; try this out in a Python shell:

```
>>> from unittest.mock import Mock
>>> m = Mock()
>>> m.any_attribute
<Mock name='mock.any_attribute' id='140716305179152'>
>>> type(m.any_attribute)
<class 'unittest.mock.Mock'>
>>> m.any_method()
<Mock name='mock.any_method()' id='140716331211856'>
>>> m.foo()
<Mock name='mock.foo()' id='140716331251600'>
>>> m.called
False
>>> m.foo.called
True
>>> m.bar.return_value = 1
>>> m.bar(42, var='thing')
1
>>> m.bar.call_args
call(42, var='thing')
```

2 Again, we're acting as if Django's `mail.outbox` didn't exist, for the sake of learning. After all, what if you were using Flask? Or what if this was an API call, not an email?

A mock is a magical object for a few reasons:

- It responds to any request for an attribute or method call with other mocks.
- You can configure it in turn to return specific values when called.
- It enables you to inspect what it was called with.

Sounds like a useful thing to be able to use in our unit tests!

Using unittest.patch

And as if that weren't enough, the `mock` module also provides a helper function called `patch`, which we can use to do the monkeypatching we did by hand earlier.

I'll explain how it all works shortly, but let's see it in action first:

src/accounts/tests/test_views.py (ch20l007)

```
from unittest import mock

from django.test import TestCase
[...]

class SendLoginEmailViewTest(TestCase):
    def test_redirects_to_home_page(self):
        [...]

    @mock.patch("accounts.views.send_mail")  ❶
    def test_sends_mail_to_address_from_post(self, mock_send_mail):  ❷
        self.client.post(
            "/accounts/send_login_email", data={"email": "edith@example.com"}
        )

        self.assertEqual(mock_send_mail.called, True)
        (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args
        self.assertEqual(subject, "Your login link for Superlists")
        self.assertEqual(from_email, "noreply@superlists")
        self.assertEqual(to_list, ["edith@example.com"])
```

❶ Here's the decorator—we'll go into detail about how it works shortly.

❷ Here's the extra argument we add to the test method. Again, detailed explanation to come, but as you'll see, it's going to do most of the work that `fake_send_mail` was doing before.

If you rerun the tests, you'll see they still pass. And because we're always suspicious of any test that still passes after a big change, let's deliberately break it just to see:

src/accounts/tests/test_views.py (ch20l008)

```
        self.assertEqual(to_list, ["schmedith@example.com"])
```

And let's add a little debug print to our view as well, to see the effects of the `mock.patch`:

src/accounts/views.py (ch20l009)

```
def send_login_email(request):
    email = request.POST["email"]
    print(type(send_mail))
    send_mail(
        [...]
```

Let's run the tests again:

```
$ python src/manage.py test accounts
[...]
....<class 'function'>
.<class 'unittest.mock.MagicMock'>
[...]
AssertionError: Lists differ: ['edith@example.com'] !=
['schmedith@example.com']
[...]

Ran 6 tests in 0.024s

FAILED (failures=1)
```

Sure enough, the tests fail. And we can see, just before the failure message, that when we print the `type` of the `send_mail` function, in the first unit test it's a normal function, but in the second unit test we're seeing a mock object.

Let's remove the deliberate mistake and dive into exactly what's going on:

src/accounts/tests/test_views.py (ch20l011)

```
@mock.patch("accounts.views.send_mail")  ❶
def test_sends_mail_to_address_from_post(self, mock_send_mail):  ❷
    self.client.post(  ❸
        "/accounts/send_login_email", data={"email": "edith@example.com"}
    )

    self.assertEqual(mock_send_mail.called, True)  ❹
    (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args  ❺
    self.assertEqual(subject, "Your login link for Superlists")
    self.assertEqual(from_email, "noreply@superlists")
    self.assertEqual(to_list, ["edith@example.com"])
```

❶ The `mock.patch()` decorator takes a dot-notation name of an object to monkeypatch. That's the equivalent of manually replacing the `send_mail` in `accounts.views`. The advantage of the decorator is that, firstly, it automatically replaces the target with a mock. And secondly, it automatically puts the original object back at the end! (Otherwise, the object stays monkeypatched for the rest of the test run, which might cause problems in other tests.)

❷ `patch` then injects the mocked object into the test as an argument to the test method. We can choose whatever name we want for it, but I usually use a convention of `mock_` plus the original name of the object.

❸ We call our view under test as usual, but everything inside this test method has our mock applied to it, so the view won't call the real `send_mail` object; it'll be seeing `mock_send_mail` instead.

❹ And we can now make assertions about what happened to that mock object during the test. We can see it was called…

❺ …and we can also unpack its various positional and keyword call arguments, to examine what it was called with. (See "On Mock call_args" on page 507 in the next chapter for a longer explanation of `.call_args`.)

All crystal clear? No? Don't worry; we'll do a couple more tests with mocks to see if they start to make more sense as we use them more.

Getting the FT a Little Further Along

First let's get back to our FT and see where it's failing:

```
$ python src/manage.py test functional_tests.test_login
[...]
AssertionError: 'Check your email' not found in 'Superlists\nEnter your email
to log in\nStart a new To-Do list'
```

Submitting the email address currently has no effect. Hmmm. Currently our form is hardcoded to send to */accounts/send_login_email*. Let's switch to using the `{% url %}` syntax just to make sure it's the right URL:

src/lists/templates/base.html (ch20l012)

```
<form method="POST" action="{% url 'send_login_email' %}">
```

Does that help? Nope, same error. Why? Ah, nothing to do with the URL actually; it's because we're not displaying a success message after we send the user an email. Let's add a test for that.

Testing the Django Messages Framework

We'll use Django's "messages framework", which is often used to display ephemeral "success" or "warning" messages to show the results of an action, something like what's shown in Figure 20-1.

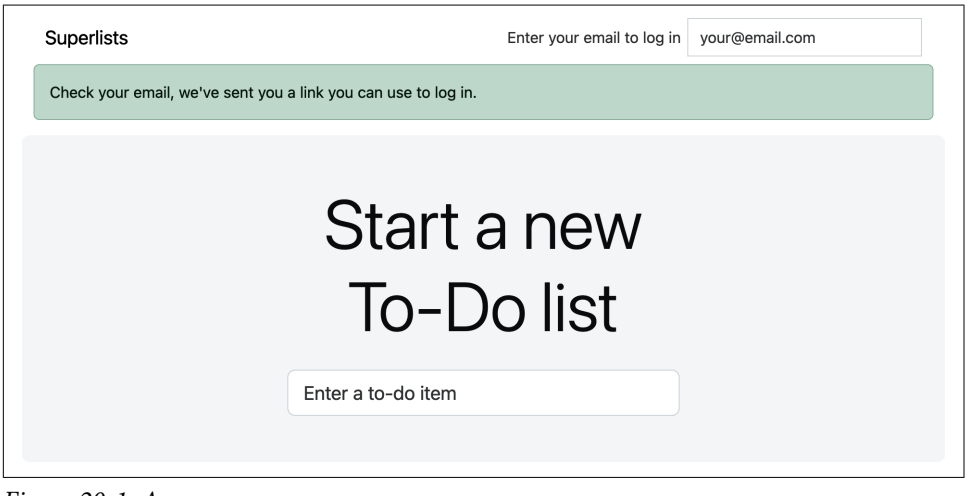


Figure 20-1. A green success message

Have a look at the Django messages docs if you haven't come across it already. Testing Django messages is a bit contorted:

src/accounts/tests/test_views.py (ch20l013)

```
def test_adds_success_message(self):
    response = self.client.post(
        "/accounts/send_login_email",
        data={"email": "edith@example.com"},
        follow=True,  ❶
    )

    message = list(response.context["messages"])[0]  ❷
    self.assertEqual(
        message.message,
        "Check your email, we've sent you a link you can use to log in.",
    )
    self.assertEqual(message.tags, "success")
```

❶ We have to pass `follow=True` to the test client to tell it to get the page *after* the 302-redirect.

❷ Then we examine the response context for a `messages` iterable, which we have to listify before it'll play nicely. (We'll use these later in a template with `{% for message in messages %}`.)

That gives:

```
$ python src/manage.py test accounts
[...]
    message = list(response.context["messages"])[0]
IndexError: list index out of range
```

And we can get it passing with:

src/accounts/views.py (ch20l014)

```
from django.contrib import messages
[...]

def send_login_email(request):
    [...]
    messages.success(
        request,
        "Check your email, we've sent you a link you can use to log in.",
    )
    return redirect("/")
```

Mocks Can Leave You Tightly Coupled to the Implementation

This sidebar is an intermediate-level testing tip. If it goes over your head the first time around, come back and take another look when you've finished this chapter.

I said testing messages is a bit contorted; it took me several goes to get it right. In fact, at a previous employer, we gave up on testing them like this and decided to just use mocks. Let's see what that would look like in this case:

src/accounts/tests/test_views.py (ch20l014-2)

```
@mock.patch("accounts.views.messages")
def test_adds_success_message_with_mocks(self, mock_messages):
    response = self.client.post(
        "/accounts/send_login_email", data={"email": "edith@example.com"}
    )

    expected = "Check your email, we've sent you a link you can use to log in."
    self.assertEqual(
        mock_messages.success.call_args,
        mock.call(response.wsgi_request, expected),
    )
```

We mock out the `messages` module, and check that `messages.success` was called with the right arguments: the original request and the message we want.

And you could get it passing by using the exact same code as earlier. Here's the problem though: the messages framework gives you more than one way to achieve the same result. I could write the code like this:

src/accounts/views.py (ch20l014-3)

```
messages.add_message(
    request,
    messages.SUCCESS,
    "Check your email, we've sent you a link you can use to log in.",
)
```

And the original, non-mocky test would still pass. But our mocky test will fail, because we're no longer calling `messages.success`; we're calling `messages.add_message`. Even though the end result is the same and our code is "correct", the test is broken.

This is what it means to say that using mocks leave you "tightly coupled with the implementation". We usually say it's better to test behaviour, not implementation details; test what happens, not how you do it. Mocks often end up erring too much on the side of the "how" rather than the "what".

Test should be about behaviour, not implementation. If your tests tie you to specific implementation details, they will prevent you from refactoring as freely.

Adding Messages to Our HTML

What happens next in the functional test? Ah. Still nothing. We need to actually add the messages to the page. Something like this:

src/lists/templates/base.html (ch20l015)

```
[...]
</nav>

{% if messages %}
  <div class="row">
    <div class="col-md-12">
      {% for message in messages %}
        {% if message.level_tag == 'success' %}
          <div class="alert alert-success">{{ message }}</div>
        {% else %}
          <div class="alert alert-warning">{{ message }}</div>
        {% endif %}
      {% endfor %}
    </div>
  </div>
{% endif %}
```

Now do we get a little further? Yes!

```
$ python src/manage.py test accounts
[...]
Ran 7 tests in 0.023s

OK

$ python src/manage.py test functional_tests.test_login
[...]
AssertionError: 'Use this link to log in' not found in 'body text tbc'
```

We need to fill out the body text of the email, with a link that the user can use to log in. Let's just cheat for now though, by changing the value in the view:

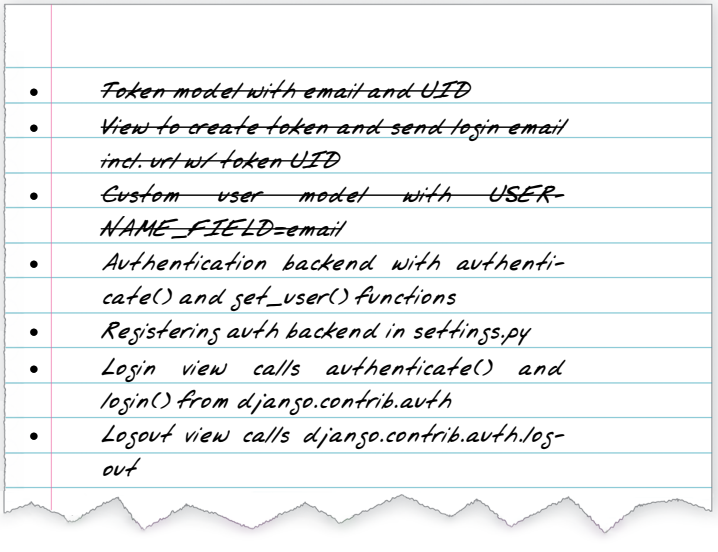
src/accounts/views.py (ch20l016)

```
    send_mail(
        "Your login link for Superlists",
        "Use this link to log in",
        "noreply@superlists",
        [email],
    )
```

That gets the FT a little further:

```
$ python src/manage.py test functional_tests.test_login
[...]
AssertionError: Could not find url in email body:
Use this link to log in
```

OK, I think we can call the `send_login_email` view done for now:



Starting on the Login URL

We're going to have to build some kind of URL! Let's build the minimal thing, just a placeholder really:

src/accounts/tests/test_views.py (ch20l017)

```
class LoginViewTest(TestCase):
    def test_redirects_to_home_page(self):
        response = self.client.get("/accounts/login?token=abcd123")
        self.assertRedirects(response, "/")
```

We're imagining we'll pass the token in as a GET parameter, after the `?`. It doesn't need to do anything for now.

I'm sure you can find your way through to getting the boilerplate in for a basic URL and view, via errors like these:

No URL:

```
AssertionError: 404 != 302 : Response didn't redirect as expected: Response
code was 404 (expected 302)
```

No view:

```
AttributeError: module 'accounts.views' has no attribute 'login'
```

Broken view:

```
ValueError: The view accounts.views.login didn't return an HttpResponse object.
It returned None instead.
```

OK!

```
$ python src/manage.py test accounts
[...]

Ran 8 tests in 0.029s
OK
```

And now we can give people a link to use. It still won't do much though, because we still don't have a token to give to the user.

Checking That We Send the User a Link with a Token

Back in our `send_login_email` view, we've tested the email subject, and the "from", and "to" fields. The body is the part that will have to include a token or URL they can use to log in. Let's spec out two tests for that:

src/accounts/tests/test_views.py (ch20l021)

```
from accounts.models import Token
[...]

class SendLoginEmailViewTest(TestCase):
    def test_redirects_to_home_page(self):
        [...]
    def test_adds_success_message(self):
        [...]
    @mock.patch("accounts.views.send_mail")
    def test_sends_mail_to_address_from_post(self, mock_send_mail):
        [...]

    def test_creates_token_associated_with_email(self):  ❶
        self.client.post(
            "/accounts/send_login_email", data={"email": "edith@example.com"}
        )
        token = Token.objects.get()
        self.assertEqual(token.email, "edith@example.com")

    @mock.patch("accounts.views.send_mail")
    def test_sends_link_to_login_using_token_uid(self, mock_send_mail):  ❷
        self.client.post(
            "/accounts/send_login_email", data={"email": "edith@example.com"}
        )

        token = Token.objects.get()
        expected_url = f"http://testserver/accounts/login?token={token.uid}"
        (subject, body, from_email, to_list), kwargs = mock_send_mail.call_args
        self.assertIn(expected_url, body)
```

❶ The first test is fairly straightforward; it checks that the token we create in the database is associated with the email address from the POST request.

❷ The second one is our second test using mocks. We mock out the `send_mail` function again using the `patch` decorator, but this time we're interested in the `body` argument from the call arguments.

Running them now will fail because we're not creating any kind of token:

```
$ python src/manage.py test accounts
[...]
accounts.models.Token.DoesNotExist: Token matching query does not exist.
[...]
accounts.models.Token.DoesNotExist: Token matching query does not exist.
```

We can get the first one to pass by creating a token:

src/accounts/views.py (ch20l022)

```
from accounts.models import Token
[...]

def send_login_email(request):
    email = request.POST["email"]
    token = Token.objects.create(email=email)
    send_mail(
        [...]
```

And now the second test prompts us to actually use the token in the body of our email:

```
[...]
AssertionError:
'http://testserver/accounts/login?token=[...]
not found in 'Use this link to log in'

FAILED (failures=1)
```

So, we can insert the token into our email like this:

src/accounts/views.py (ch20l023)

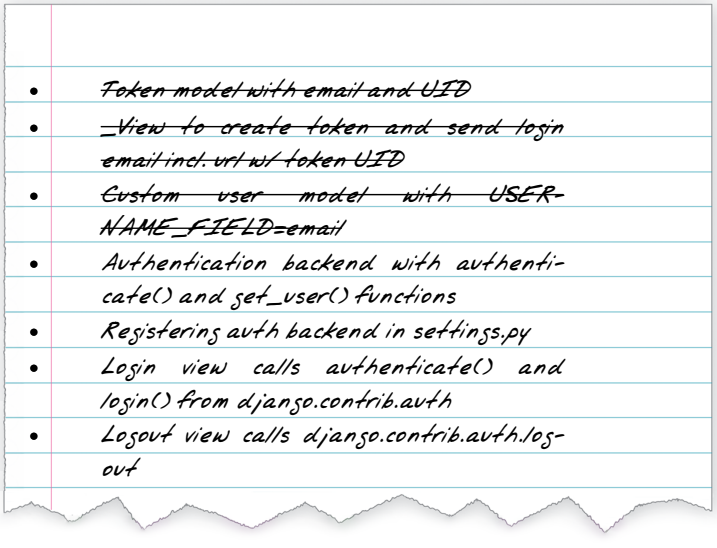
```
from django.urls import reverse
[...]

def send_login_email(request):
    email = request.POST["email"]
    token = Token.objects.create(email=email)
    url = request.build_absolute_uri(  ❶
        reverse("login") + "?token=" + str(token.uid),
    )
    message_body = f"Use this link to log in:\n\n{url}"
    send_mail(
        "Your login link for Superlists",
        message_body,
        "noreply@superlists",
        [email],
    )
    [...]
```

❶ `request.build_absolute_uri` deserves a mention—it's one way to build a "full" URL, including the domain name and the HTTP(S) part, in Django. There are other ways, but they usually involve getting into the "sites" framework, which gets complicated pretty quickly. You can find lots more discussion on this if you're curious by doing a bit of googling.

And the tests pass:

```
OK
```

I think *that's* our `send_login_email` view done:



The next piece in the puzzle is the authentication backend, whose job it will be to examine tokens for validity and then return the corresponding users. Then, we need to get our login view to actually log users in, if they can authenticate.

De-Spiking Our Custom Authentication Backend

Here's how our authentication backend looked in the spike:

```
class PasswordlessAuthenticationBackend(BaseBackend):
    def authenticate(self, request, uid):
        print("uid", uid, file=sys.stderr)
        if not Token.objects.filter(uid=uid).exists():
            print("no token found", file=sys.stderr)
            return None
        token = Token.objects.get(uid=uid)
        print("got token", file=sys.stderr)
        try:
            user = ListUser.objects.get(email=token.email)
            print("got user", file=sys.stderr)
            return user
        except ListUser.DoesNotExist:
            print("new user", file=sys.stderr)
            return ListUser.objects.create(email=token.email)

    def get_user(self, email):
        return ListUser.objects.get(email=email)
```

Decoding this:

- We take a UID and check if it exists in the database.
- We return `None` if it doesn't.
- If it does exist, we extract an email address, and either find an existing user with that address or create a new one.

One if = One More Test

A rule of thumb for these sorts of tests: any `if` means an extra test, and any `try/except` means an extra test. So, this should be about three tests. How about something like this?

src/accounts/tests/test_authentication.py (ch20l024)

```
from django.http import HttpRequest
from django.test import TestCase

from accounts.authentication import PasswordlessAuthenticationBackend
from accounts.models import Token, User


class AuthenticateTest(TestCase):
    def test_returns_None_if_no_such_token(self):
        result = PasswordlessAuthenticationBackend().authenticate(
            HttpRequest(), "no-such-token"
        )
        self.assertIsNone(result)

    def test_returns_new_user_with_correct_email_if_token_exists(self):
        email = "edith@example.com"
        token = Token.objects.create(email=email)
        user = PasswordlessAuthenticationBackend().authenticate(
            HttpRequest(), token.uid
        )
        new_user = User.objects.get(email=email)
        self.assertEqual(user, new_user)

    def test_returns_existing_user_with_correct_email_if_token_exists(self):
        email = "edith@example.com"
        existing_user = User.objects.create(email=email)
        token = Token.objects.create(email=email)
        user = PasswordlessAuthenticationBackend().authenticate(
            HttpRequest(), token.uid
        )
        self.assertEqual(user, existing_user)
```

In *authenticate.py*, we'll just have a little placeholder:

src/accounts/authentication.py (ch20l025)

```
class PasswordlessAuthenticationBackend:
    def authenticate(self, request, uid):
        pass
```

How do we get on?

```
$ python src/manage.py test accounts

.FE.........
======================================================================
ERROR: test_returns_new_user_with_correct_email_if_token_exists (accounts.tests
.test_authentication.AuthenticateTest.test_returns_new_user_with_correct_email_
if_token_exists)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/accounts/tests/test_authentication.py", line 21, in
test_returns_new_user_with_correct_email_if_token_exists
    new_user = User.objects.get(email=email)
[...]
accounts.models.User.DoesNotExist: User matching query does not exist.


======================================================================
FAIL: test_returns_existing_user_with_correct_email_if_token_exists (accounts.t
ests.test_authentication.AuthenticateTest.test_returns_existing_user_with_corre
ct_email_if_token_exists)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/accounts/tests/test_authentication.py", line 31, in
test_returns_existing_user_with_correct_email_if_token_exists
    self.assertEqual(user, existing_user)
    ~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^^^^^^^
AssertionError: None != <User: User object (edith@example.com)>

 ----------------------------------------------------------------------
Ran 13 tests in 0.038s

FAILED (failures=1, errors=1)
```

Here's a first cut:

src/accounts/authentication.py (ch20l026)

```
from accounts.models import Token, User


class PasswordlessAuthenticationBackend:
    def authenticate(self, request, uid):
        token = Token.objects.get(uid=uid)
        return User.objects.get(email=token.email)
```

Now, instead of one `FAIL` and one `ERROR`, we get two `ERROR`s:

```
$ python src/manage.py test accounts

ERROR: test_returns_None_if_no_such_token (accounts.tests.test_authentication.A
uthenticateTest.test_returns_None_if_no_such_token)
[...]
accounts.models.Token.DoesNotExist: Token matching query does not exist.

ERROR: test_returns_new_user_with_correct_email_if_token_exists (accounts.tests
.test_authentication.AuthenticateTest.test_returns_new_user_with_correct_email_
if_token_exists)
[...]
accounts.models.User.DoesNotExist: User matching query does not exist.
```

Notice that our third test, `test_returns_existing_user_with_correct_email_if_token_exists`, is actually passing. Our code *does* currently handle the "happy path", where both the token and the user already exist in the database.

Let's fix each of the remaining ones in turn. Notice how the test names are telling us exactly what we need to do. First, `test_returns_None_if_no_such_token`, which is telling us what to do if the token doesn't exist:

src/accounts/authentication.py (ch20l027)

```
    def authenticate(self, request, uid):
        try:
            token = Token.objects.get(uid=uid)
            return User.objects.get(email=token.email)
        except Token.DoesNotExist:
            return None
```

That gets us down to one failure:

```
ERROR: test_returns_new_user_with_correct_email_if_token_exists (accounts.tests
.test_authentication.AuthenticateTest.test_returns_new_user_with_correct_email_
if_token_exists)
[...]
accounts.models.User.DoesNotExist: User matching query does not exist.

FAILED (errors=1)
```

OK, so we need to return a `new_user_with_correct_email` `if_token_exists`? We can do that!

src/accounts/authentication.py (ch20l028)

```
    def authenticate(self, request, uid):
        try:
            token = Token.objects.get(uid=uid)
            return User.objects.get(email=token.email)
        except User.DoesNotExist:
            return User.objects.create(email=token.email)
        except Token.DoesNotExist:
            return None
```

That's turned out neater than our spike!

The get_user Method

We've handled the `authenticate` function, which Django will use to log new users in. The second part of the protocol we have to implement is the `get_user` method, whose job is to retrieve a user based on their unique identifier (the email address), or to return `None` if it can't find one. (Have another look at the spiked code if you need a reminder.)

Here are a couple of tests for those two requirements:

src/accounts/tests/test_authentication.py (ch20l030)

```
class GetUserTest(TestCase):
    def test_gets_user_by_email(self):
        User.objects.create(email="another@example.com")
        desired_user = User.objects.create(email="edith@example.com")
        found_user = PasswordlessAuthenticationBackend().get_user("edith@example.com")
        self.assertEqual(found_user, desired_user)

    def test_returns_None_if_no_user_with_that_email(self):
        self.assertIsNone(
            PasswordlessAuthenticationBackend().get_user("edith@example.com")
        )
```

And our first failure:

```
AttributeError: 'PasswordlessAuthenticationBackend' object has no attribute
'get_user'
```

Let's create a placeholder one then:

src/accounts/authentication.py (ch20l031)

```
class PasswordlessAuthenticationBackend:
    def authenticate(self, request, uid):
        [...]

    def get_user(self, email):
        pass
```

Now we get:

```
    self.assertEqual(found_user, desired_user)
AssertionError: None != <User: User object (edith@example.com)>
```

And (step by step, just to see if our test fails the way we think it will):

src/accounts/authentication.py (ch20l033)

```
    def get_user(self, email):
        return User.objects.first()
```

That gets us past the first assertion, and onto:

```
    self.assertEqual(found_user, desired_user)
AssertionError: <User: User object (another@example.com)> != <User: User object
(edith@example.com)>
```

And so, we call `get` with the email as an argument:

src/accounts/authentication.py (ch20l034)

```
    def get_user(self, email):
        return User.objects.get(email=email)
```

Now our test for the `None` case fails:

```
ERROR: test_returns_None_if_no_user_with_that_email (accounts.tests.test_authen
tication.GetUserTest.test_returns_None_if_no_user_with_that_email)
[...]
accounts.models.User.DoesNotExist: User matching query does not exist.
```

That prompts us to finish the method like this:

src/accounts/authentication.py (ch20l035)

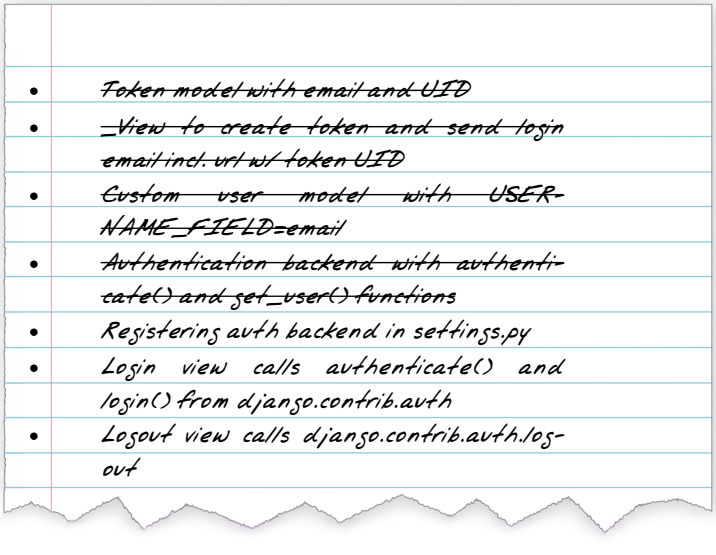
```
    def get_user(self, email):
        try:
            return User.objects.get(email=email)
        except User.DoesNotExist:
            return None  ❶
```

❶ You could just use `pass` here, and the function would return `None` by default. However, because we specifically need the function to return `None`, the "explicit is better than implicit" rule applies here.

That gets us to passing tests:

```
OK
```

And we have a working authentication backend!



Let's call that a win and, in the next chapter, we'll work on integrating it into our login view and getting our FT passing.

On Mocking in Python

Mocking and external dependencies

One place to consider using mocking is when we have an external dependency that we don't want to actually use in our tests. A mock can be used to simulate the third-party API. Whilst it is possible to "roll your own" mocks in Python, a mocking framework like the `unittest.mock` module provides a lot of helpful shortcuts that will make it easier to write (and more importantly, read) your tests.

The mock library

The `unittest.mock` module from Python's standard library contains most everything you might need for monkeypatching and mocking in Python.[3]

Monkeypatching

This is the process of replacing an object in a namespace at runtime. We use it in our unit tests to replace a real function that has undesirable side effects with a mock object, using the `mock.patch` decorator.

The mock.patch decorator

`unittest.mock` provides a function called `patch`, which can be used to "mock out" (monkeypatch) any object from the module you're testing. It's commonly used as a decorator on a test method. Importantly, it "undoes" the mocking at the end of the test for you, to avoid contamination between tests.

Mocks can leave you tightly coupled to the implementation

As discussed in the earlier sidebar, mocks can leave you tightly coupled to your implementation. For that reason, you shouldn't use them unless you have a good reason.

3 This library was originally written as a standalone package by Michael Foord while he was working at the company that later spawned PythonAnywhere, a few years before I joined. It became part of the standard library in Python 3.3. Michael was a friend, and sadly passed away in 2025.

CHAPTER 21

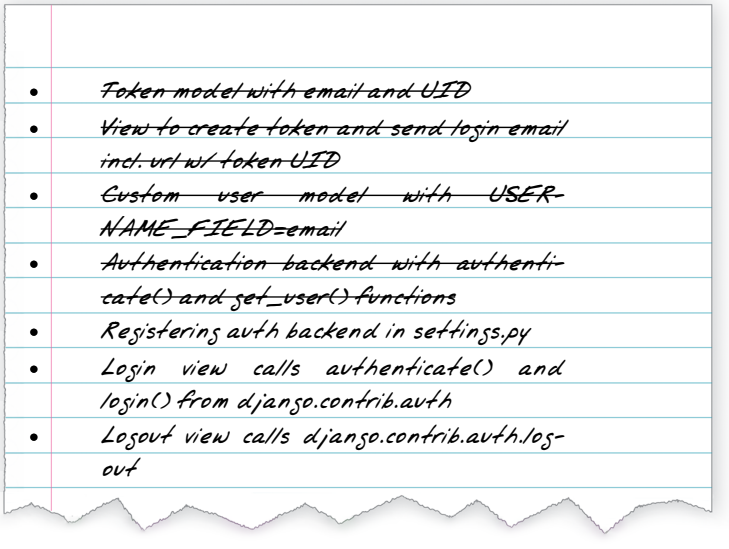
Using Mocks for Test Isolation

In this chapter, we'll finish up our login system. While doing so, we'll explore an alternative use of mocks: to isolate parts of the system from each other. This enables more targeted testing, fights combinatorial explosion, and reduces duplication between tests.

In this chapter, we start to drift towards what's called "London-school TDD", which is a variant on the "Classical" or "Detroit" style of TDD that I mostly show in the book. We won't get into the details here, but London-school TDD places more emphasis on mocking and isolating parts of the system. As always, there are pros and cons! Read more at Online Appendix: Test Isolation and "Listening to Your Tests".

Along the way, we'll learn a few more useful features of `unittest.mock`, and we'll also have a discussion about how many tests are "enough".

Using Our Auth Backend in the Login View

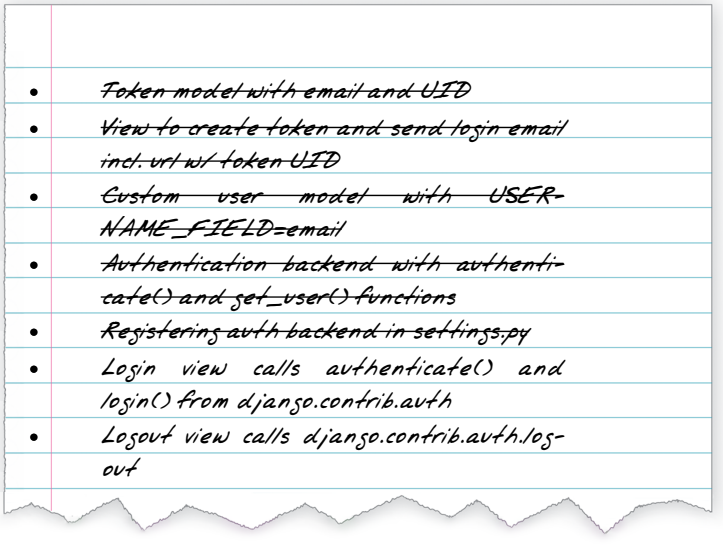


We got our auth backend ready in the last chapter; now we need use the backend in our login view. But first, as our scratchpad says, we need to add it to *settings.py*:

src/superlists/settings.py (ch21l001)

```
AUTH_USER_MODEL = "accounts.User"
AUTHENTICATION_BACKENDS = [
    "accounts.authentication.PasswordlessAuthenticationBackend",
]

[...]
```

That was easy!



Next, let's write some tests for what should happen in our view. Looking back at the spike again:

src/accounts/views.py

```
def login(request):
    print("login view", file=sys.stderr)
    uid = request.GET.get("uid")
    user = auth.authenticate(uid=uid)
    if user is not None:
        auth.login(request, user)
    return redirect("/")
```

You can view the contents of files from the spike using, for example, `git show passwordless-spike:src/accounts/views.py`.

We call `django.contrib.auth.authenticate` and then, if it returns a user, we call `django.contrib.auth.login`.

This is a good time to check out the Django docs on authentication for a little more context.

Straightforward Non-Mocky Test for Our View

Here's the most obvious test we might want to write, thinking in terms of the *behaviour* we want:

- If someone has a valid token, they should get logged in.
- If someone tries to use an invalid token (or does not have one), it should not log them in.

Here's how we might add the happy-path test for the user with the valid token:

src/accounts/tests/test_views.py (ch21l002)

```
from django.contrib import auth
[...]

class LoginViewTest(TestCase):
    def test_redirects_to_home_page(self):
        [...]

    def test_logs_in_if_given_valid_token(self):
        anon_user = auth.get_user(self.client)  ❶
        self.assertEqual(anon_user.is_authenticated, False)  ❷

        token = Token.objects.create(email="edith@example.com")
        self.client.get(f"/accounts/login?token={token.uid}")

        user = auth.get_user(self.client)
        self.assertEqual(user.is_authenticated, True)  ❸
        self.assertEqual(user.email, "edith@example.com")  ❸
```

❶ We use Django's `auth.get_user()` to extract the current user from the test client.

❷ We verify we're not logged in before we start. (This isn't strictly necessary, but it's always nice to know you're on firm ground.)

❸ And here's where we check that we've been logged in, with a user with the right email address.

And that will fail as expected:

```
    self.assertEqual(user.is_authenticated, True)
AssertionError: False != True
```

We can get it to pass by "cheating", like this:

src/accounts/views.py (ch21l003)

```
from django.contrib import auth, messages
[...]
from accounts.models import Token, User


def send_login_email(request):
    [...]


def login(request):
    user = User.objects.create(email="edith@example.com")
    auth.login(request, user)
    return redirect("/")
```

```
...

    OK
```

That forces us to write another test:

src/accounts/tests/test_views.py (ch21l004)

```
def test_shows_login_error_if_token_invalid(self):
    response = self.client.get("/accounts/login?token=invalid-token", follow=True)
    user = auth.get_user(self.client)
    self.assertEqual(user.is_authenticated, False)
    message = list(response.context["messages"])[0]
    self.assertEqual(
        message.message,
        "Invalid login link, please request a new one",
    )
    self.assertEqual(message.tags, "error")
```

And now we get that passing by using the most straightforward implementation…

src/accounts/views.py (ch21l005)

```
def login(request):
    if Token.objects.filter(uid=request.GET["token"]).exists():  ❶
        user = User.objects.create(email="edith@example.com")  ❷ ❸
        auth.login(request, user)
    else:
        messages.error(request, "Invalid login link, please request a new one")  ❹
    return redirect("/")
```

❶ Oh wait; we forgot about our authentication backend and just did the query directly from the token model! Well that's arguably more straightforward, but how do we force ourselves to write the code the way we want to—i.e., using Django's authentication API?

❷ Oh dear, and the email address is still hardcoded. We might have to think about writing an extra test to force ourselves to fix that.

❸ Oh—also, we're hardcoding the creation of a user every time, but actually, we want to have the get-or-create logic that we implemented in our backend.

❹ This bit is OK at least!

Is this starting to feel a bit familiar? We've already written all the tests for the various permutations of our authentication logic, and we're considering writing equivalent tests at the views layer.

Combinatorial Explosion

Table 21-1 recaps the tests we might want to write at each layer in our application.

Table 21-1. What we want to test in each layer

Views layer	Authentication backend	Models layer
• Valid token means user is logged in • Invalid token means user is not logged in	• Returns correct existing user for a valid token • Creates a new user for a new email address • Returns `none` for an invalid token	• Token associates email and UID • User can be retrieved from token UID

We already have three tests in the models layer, and five in the authentication layer. We started off writing the tests in the views layer, where—*conceptually*—we only really want two test cases, and we're finding ourselves wondering if we need to write a whole bunch of tests that essentially duplicate the authentication layer tests. This is an example of the *combinatorial explosion* problem.

The Car Factory Example

Imagine we're testing a car factory:

- First, we choose the car type: normal, station-wagon, or convertible.
- Then, we choose the engine type: petrol, diesel, or electric.
- Finally, we choose the colour: red, white, or hot pink.

Here's how it might look in code:

```
def build_car(car_type, engine_type, colour):
    engine = _create_engine(engine_type)
    naked_car = _assemble_car(engine, car_type)
    finished_car = _paint_car(naked_car, colour)
    return finished_car
```

How many tests do we need? Well, the upper bound to test every possible combination is $3 \times 3 \times 3 = 27$ tests. That's a lot!

How many tests do we *actually* need to write? Well, it depends on how we're testing, how the different parts of the factory are integrated, and what we know about the system. Do we need to test every single colour? Maybe! Or, maybe, if we're happy that we can do two different colours, then we're happy that we can do any number—whether it's two, three, or hundreds. Perhaps we need two tests, maybe three.

OK, but do we need to test that painting works for all the different engine types? Well, the painting process is probably independent of engine type: if we can paint a diesel in red, we can paint it in pink or white too.

But, perhaps it *is* affected by the car type: painting a convertible with a fabric roof might be a very different technological process to painting a hard-bodied car. So, we'd probably want to test that painting *in general* works for each car type (three tests), but we don't need to test that painting works for every engine type.

What we're analysing here is the level of "coupling" between the different parts of the system. Painting is tightly coupled to car type, but not to engine type. Painting "needs to know" about car types, but it does not "need to know" about engine types.

The more tightly coupled two parts of the system are, the more tests you'll need to write to cover all the combinations of their behaviour.

Another way of thinking about it is: what level are we writing tests at? You can choose to write low-level tests that cover only one part of the assembly process, or higher-level ones that test several steps together—or perhaps all of them end-to-end. See Figure 21-1.

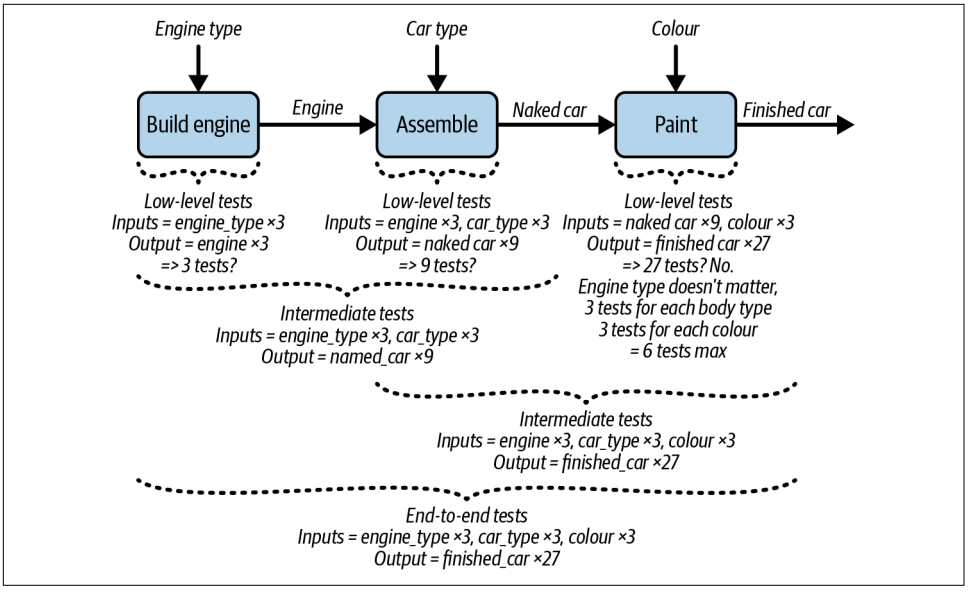


Figure 21-1. Analysing how many tests are needed at different levels

Analysing things in these terms, we think about the inputs and outputs that apply to each type of test, as well as which attributes of the inputs matter, and which don't.

Testing the first stage of the process—building the engine—is straightforward. The "engine type" input has three possible values, so we need three tests of the output, which is the engine. If we're testing at the end-to-end level, no matter how many tests we have in total, we know we'll need at least three to be the tests that check if we can produce a car with a working engine of each type.

Testing the painting needs a bit more thought. If we test at the low level, the inputs are a naked car and a paint colour. There are theoretically nine types of naked car; do we need to test all of them? No. The engine type doesn't matter; we only need to test one of each body type. Does that mean $3 \times 3 = 9$ tests? No. The colour and body type are independent. We can just test that all three colours work, and that all three body types work—so that's six tests.

What about at the end-to-end level? It depends if we're being rigorous about "closed-box" testing, where we're not supposed to know anything about how the production process works. In that case, maybe we *do* need 27 tests. But if we allow that we know about the internals, then we can apply similar reasoning to what we used at the lower level. However many tests we end up with, we need three of them to be checking each colour, and three that check that each body type can be painted.

Let's see if we can apply this sort of analysis to our authentication system.

Using Mocks to Test Parts of Our System in Isolation

To recap, so far we have some minimal tests at the models layer, and we have comprehensive tests of our authentication backend, and we're now wondering how many tests we need at the views layer.

Here's the current state of our view:

src/accounts/views.py

```
def login(request):
    if Token.objects.filter(uid=request.GET["token"]).exists():
        user = User.objects.create(email="edith@example.com")
        auth.login(request, user)
    else:
        messages.error(request, "Invalid login link, please request a new one")
    return redirect("/")
```

We know we want to transform it to something like this:

src/accounts/views.py

```
def login(request):
    if user := auth.authenticate(uid=request.GET.get("token"))  ❶
        auth.login(request, user)  ❷
    else:
        messages.error(request, "Invalid login link, please request a new one")  ❸

    return redirect("/")
```

❶ We want to refactor our logic to use the `authenticate()` function from our backend. Really good place for a walrus (`:=`) too!

❷ We have the happy path where the user gets logged in.

❸ We have the unhappy path where the user gets an error message instead.

But currently, our tests are letting us "get away" with the wrong implementation. Here are three possible options for getting ourselves to the right state:

1. Add more tests for all possible combinations at the view level (token exists but no user, token exists for an existing user, invalid token, etc.), until we end up duplicating all the logic in the auth backend in our view—and then feel justified in refactoring across to just calling the auth backend.
2. Stick with our current two tests, and decide it's OK to refactor already.
3. Test the view in isolation, using mocks to verify that we call the auth backend.

Each option has pros and cons! If I was going for option (1), essentially going all in on test coverage at the views layer, I'd probably think about deleting all the tests at the auth layer afterwards.

If you were to ask me what my personal preference or instinctive choice would be, I'd say at this point it might be to go with (2), and say with one happy-path and one unhappy-path test, we're OK to refactor and switch across already.

But because this chapter is about mocks, let's investigate option (3) instead. Besides, it'll be an excuse to do fun things with them, like playing with `.return_value`.

So far, we've used mocks to test external dependencies, like Django's mail-sending function. The main reason to use a mock we've discussed so far is to isolate ourselves from external side effects—in this case, to avoid sending out actual emails during our tests.

In this section, we'll look at a different possible use case for mocks: testing parts of our *own* code in isolation from each other, as a way of reducing duplication and avoiding combinatorial explosion in our tests.

Mocks Can Also Let You Test the Implementation, When It Matters

On top of that, the fact that we're using the Django `auth.authenticate` function rather than calling our own code directly is relevant. Django has already introduced an abstraction: to decouple the specifics of authentication backends from the views that use them. This makes it easier for us to add further backends in future.

So in this case (in contrast to the example in "Mocks Can Leave You Tightly Coupled to the Implementation" on page 481) the implementation *does* matter, because we've decided to use a particular, specific interface to implement our authentication system. This is something we might want to document and verify in our tests—and mocks are one way to enable that.

Starting Again: Test-Driving Our Implementation with Mocks

Let's see how things would look if we had decided to test-drive our implementation with mocks in the first place. We'll start by reverting all the authentication stuff, both from our test and from our view.

Let's disable the test first (we can re-enable them later to sense-check things):

src/accounts/tests/test_views.py (ch21l006)

```
class LoginViewTest(TestCase):
    def test_redirects_to_home_page(self):  ❶
        [...]
    def DONT_test_logs_in_if_given_valid_token(self):  ❷
        [...]
    def DONT_test_shows_login_error_if_token_invalid(self):  ❷
        [...]
```

❶ We can leave the test for the redirect, as that doesn't involve the auth framework.

❷ We change the test name so it no longer starts with `test_`, using a highly noticeable set of capital letters so we don't forget to come back and re-enable them later. I call this "DONTifying" tests. :)

Now let's revert the view, and replace our hacky code with some to-dos:

src/accounts/views.py (ch21l007)

```
# from django.contrib import auth, messages  ❶
from django.contrib import messages
[...]


def login(request):
    # TODO: call authenticate(),  ❷
    # then auth.login() with the user if we get one,
    # or messages.error() if we get None.
    return redirect("/")
```

❶ In order to demonstrate a common error message shortly, I'm also reverting our import of the `contrib.auth` module.

❷ And here's where we delete our first implementation and replace it with some to-dos.

Let's check that all our tests pass:

```
$ python src/manage.py test accounts
[...]
Ran 15 tests in 0.021s

OK
```

Now let's start again with mock-based tests. First, we can write a test that makes sure we call `authenticate()` correctly:

src/accounts/tests/test_views.py (ch21l008)

```
class LoginViewTest(TestCase):
    [...]

    @mock.patch("accounts.views.auth")  ❶
    def test_calls_authenticate_with_uid_from_get_request(self, mock_auth):  ❷
        self.client.get("/accounts/login?token=abcd123")
        self.assertEqual(
            mock_auth.authenticate.call_args,  ❸
            mock.call(uid="abcd123"),  ❹
        )
```

❶ We expect to be using the `django.contrib.auth` module in *views.py*, and we mock it out here. Note that this time, we're not mocking out a function; we're mocking out a whole module, and thus implicitly mocking out all the functions (and any other objects) that module contains.

❷ As usual, the mocked object is injected into our test method.

❸ This time, we've mocked out a module rather than a function. So we examine the `call_args`—not of the `mock_auth` module, but of the `mock_auth.authenticate` function. Because all the attributes of a mock are more mocks, that's a mock too. You can start to see why `Mock` objects are so convenient, compared to trying to build your own.

❹ Now, instead of "unpacking" the call args, we use the `call` function for a neater way of saying what it should have been called with—that is, the token from the GET request. (See "On Mock call_args" on page 507.)

On Mock call_args

The `.call_args` property on a mock represents the positional and keyword arguments that the mock was called with. It's a special "call" object type, which is essentially a tuple of `(positional_args, keyword_args)`. `positional_args` is itself a tuple, consisting of the set of positional arguments. `keyword_args` is a dictionary. Here they all are in action:

```
>>> from unittest.mock import Mock, call
>>> m = Mock()
>>> m(42, 43, 'positional arg 3', key='val', thing=666)
<Mock name='mock()' id='139909729163528'>

>>> m.call_args
call(42, 43, 'positional arg 3', key='val', thing=666)

>>> m.call_args == ((42, 43, 'positional arg 3'), {'key': 'val', 'thing': 666})
True
>>> m.call_args == call(42, 43, 'positional arg 3', key='val', thing=666)
True
```

So in our test, we could have done this instead:

src/accounts/tests/test_views.py

```
        self.assertEqual(
            mock_auth.authenticate.call_args,
            ((,), {'uid': 'abcd123'})
        )
        # or this
        args, kwargs = mock_auth.authenticate.call_args
        self.assertEqual(args, (,))
        self.assertEqual(kwargs, {'uid': 'abcd123'})
```

But you can see how using the `call` helper is nicer.

See also "Avoid Mock's Magic assert_called…Methods?" on page 510, for some discussion of `call_args` versus the magic `assert_called_with` methods.

What happens when we run the test? The first error is this:

```
$ python src/manage.py test accounts
[...]
AttributeError: <module 'accounts.views' from
'...goat-book/src/accounts/views.py'> does not have the attribute 'auth'
```

`module foo does not have the attribute bar` is a common first failure in a test that uses mocks. It's telling you that you're trying to mock out something that doesn't yet exist (or isn't yet imported) in the target module.

Once we reimport `django.contrib.auth`, the error changes:

src/accounts/views.py (ch21l009)

```
from django.contrib import auth, messages
[...]
```

Now we get:

```
FAIL: test_calls_authenticate_with_uid_from_get_request [...]
[...]
AssertionError: None != call(uid='abcd123')
```

It's telling us that the view doesn't call the `auth.authenticate` function at all. Let's fix that, but get it deliberately wrong, just to see:

src/accounts/views.py (ch21l010)

```
def login(request):
    # TODO: call authenticate(),
    auth.authenticate("bang!")
    # then auth.login() with the user if we get one,
    # or messages.error() if we get None.
    return redirect("/")
```

Bang, indeed!

```
$ python src/manage.py test accounts
[...]
AssertionError: call('bang!') != call(uid='abcd123')
[...]
FAILED (failures=1)
```

Let's give `authenticate` the arguments it expects then:

src/accounts/views.py (ch21l011)

```
def login(request):
    # TODO: call authenticate(),
    auth.authenticate(uid=request.GET["token"])
    # then auth.login() with the user if we get one,
    # or messages.error() if we get None.
    return redirect("/")
```

That gets us to passing tests:

```
$ python src/manage.py test accounts
Ran 16 tests in 0.023s

OK
```

Using mock.return_value

Next, we want to check that if the authenticate function returns a user, we pass that into `auth.login`. Let's see how that test looks:

src/accounts/tests/test_views.py (ch21l012)

```
@mock.patch("accounts.views.auth")  ❶
def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth):
    response = self.client.get("/accounts/login?token=abcd123")
    self.assertEqual(
        mock_auth.login.call_args,  ❷
        mock.call(
            response.wsgi_request,  ❸
            mock_auth.authenticate.return_value,  ❹
        ),
    )
```

❶ We mock the `contrib.auth` module again.

❷ This time we examine the call args for the `auth.login` function.

❸ We check that it's called with the request object that the view sees…

❹ …and we check that the second argument was "whatever the `authenticate()` function returned". Because `authenticate()` is also mocked out, we can use its special `.return_value` attribute. We know that, in real life, that will be a user object. But in this test, it's all mocks. Can you see what I mean about mocky tests being hard to understand sometimes?

When you call a mock, you get another mock. But you can also get a copy of that returned mock from the original mock that you called. Boy, it sure is hard to explain this stuff without saying "mock" a lot! Another little console illustration might help:

```
>>> m = Mock()
>>> thing = m()
>>> thing
<Mock name='mock()' id='140652722034952'>
>>> m.return_value
<Mock name='mock()' id='140652722034952'>
>>> thing == m.return_value
True
```

Avoid Mock's Magic assert_called...Methods?

If you've used `unittest.mock` before, you may have come across its special `assert_called...` methods, and you may be wondering why I didn't use them.

For example, instead of doing:

```
self.assertEqual(a_mock.call_args, call(foo, bar))
```

You can just do:

```
a_mock.assert_called_with(foo, bar)
```

And the *mock* library will raise an `AssertionError` for you if there is a mismatch.

Why not use that? For me, the problem with these magic methods is that it's too easy to make a silly typo and end up with a test that always passes:

```
a_mock.asssert_called_with(foo, bar)  # will always pass
```

Unless you get the magic method name exactly right,[1] it will just silently return another mock, and you may not realise that you've written a test that tests nothing at all. That's why I prefer to always have an explicit `unittest` method in there.[2]

In any case, what do we get from running the test?

```
$ python src/manage.py test accounts
[...]
AssertionError: None != call(<WSGIRequest: GET '/accounts/login?t[...]
```

Sure enough, it's telling us that we're not calling `auth.login()` at all yet. Let's first try doing that deliberately wrong as usual!

src/accounts/views.py (ch21l013)

```
def login(request):
    # TODO: call authenticate(),
    auth.authenticate(uid=request.GET["token"])
    # then auth.login() with the user if we get one,
    auth.login("ack!")
    # or messages.error() if we get None.
    return redirect("/")
```

1 There was actually an attempt to mitigate this problem in Python 3.5, with the addition of an `unsafe` argument that defaults to `False`, which will cause the mock to raise `AttributeError` for some common misspellings of `assert_`. Just not, for example, the one I'm using here—so I prefer not to rely on that. More info in the Python docs.

2 If you're using Pytest, there's an additional benefit to seeing the `assert` keyword rather than a normal method call: it makes the assert pop out.

Ack, indeed!

```
$ python src/manage.py test accounts
[...]

ERROR: test_redirects_to_home_page
[...]
TypeError: login() missing 1 required positional argument: 'user'

FAIL: test_calls_auth_login_with_user_if_there_is_one [...]
[...]
AssertionError: call('ack!') != call(<WSGIRequest: GET
'/accounts/login?token=[...]
[...]

Ran 17 tests in 0.026s

FAILED (failures=1, errors=1)
```

That's one expected failure from our mocky test, and one (more) unexpected failure from the non-mocky test.

Let's see if we can fix them:

src/accounts/views.py (ch21l014)

```
def login(request):
    # TODO: call authenticate(),
    user = auth.authenticate(uid=request.GET["token"])
    # then auth.login() with the user if we get one,
    auth.login(request, user)
    # or messages.error() if we get None.
    return redirect("/")
```

Well, that does fix our mocky test, but not the other one; it now has a slightly different complaint:

```
ERROR: test_redirects_to_home_page
(accounts.tests.test_views.LoginViewTest.test_redirects_to_home_page)
[...]
  File "...goat-book/src/accounts/views.py", line 33, in login
    auth.login(request, user)
[...]
AttributeError: 'AnonymousUser' object has no attribute '_meta'
```

It's because we're still calling `auth.login` indiscriminately on any kind of user, and that's causing problems back in our original test for the redirect, which *isn't* currently mocking out `auth.login`.

We can get back to passing like this:

src/accounts/views.py (ch21l015)

```
def login(request):
    # TODO: call authenticate(),
    if user := auth.authenticate(uid=request.GET["token"]):
        # then auth.login() with the user if we get one,
        auth.login(request, user)
```

This gets our unit test passing:

```
$ python src/manage.py test accounts
[...]

OK
```

Using .return_value During Test Setup

I'm a little nervous that we've introduced an `if` without an *explicit* test for it. Testing the unhappy path will reassure me. We can use our existing test for the error case to crib from.

We want to be able to set up our mocks to say: `auth.authenticate()` should return `None`. We can do that by setting the `.return_value` on the mock:

src/accounts/tests/test_views.py (ch21l016)

```
@mock.patch("accounts.views.auth")
def test_adds_error_message_if_auth_user_is_None(self, mock_auth):
    mock_auth.authenticate.return_value = None  ❶

    response = self.client.get("/accounts/login?token=abcd123", follow=True)

    message = list(response.context["messages"])[0]
    self.assertEqual(  ❷
        message.message,
        "Invalid login link, please request a new one",
    )
    self.assertEqual(message.tags, "error")
```

❶ We use `.return_value` on our mock once again. But this time, we assign to it *before* it's used (in the setup part of the test—aka the "arrange" or "given" phase), rather than reading from it (in the assert/"when" part), as we did earlier.

❷ Our asserts are copied across from the existing test for the error case, `DONT_test_shows_login_error_if_token_invalid()`.

That gives us this somewhat cryptic, but expected failure:

```
ERROR: test_adds_error_message_if_auth_user_is_None [...]
[...]
    message = list(response.context["messages"])[0]
              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^^^
IndexError: list index out of range
```

Essentially, that's saying there are no messages in our response. We can get it passing like this, starting with a deliberate mistake as always:

src/accounts/views.py (ch21l017)

```
def login(request):
    # TODO: call authenticate(),
    if user := auth.authenticate(uid=request.GET["token"]):
        # then auth.login() with the user if we get one,
        auth.login(request, user)
    else:
        # or messages.error() if we get None.
        messages.error(request, "boo")
    return redirect("/")
```

Which gives us:

```
AssertionError: 'boo' != 'Invalid login link, please request a new one'
```

And so:

src/accounts/views.py (ch21l018)

```
def login(request):
    # TODO: call authenticate(),
    if user := auth.authenticate(uid=request.GET["token"]):
        # then auth.login() with the user if we get one,
        auth.login(request, user)
    else:
        # or messages.error() if we get None.
        messages.error(request, "Invalid login link, please request a new one")
    return redirect("/")
```

Now our tests pass:

```
$ python src/manage.py test accounts
[...]

Ran 18 tests in 0.025s

OK
```

And we can do a final refactor to remove those comments:

src/accounts/views.py (ch21l019)

```
from accounts.models import Token  ❶
[...]


def login(request):  ❷
    if user := auth.authenticate(uid=request.GET["token"]):
        auth.login(request, user)
    else:
        messages.error(request, "Invalid login link, please request a new one")
    return redirect("/")
```

❶ We no longer need to explicitly import the user model

❷ and our view is down to just five lines.

Lovely! What's next?

UnDONTifying

Remember we still have the DONTified, non-mocky tests? Let's re-enable now to sense-check that our mocky tests have driven us to the right place:

src/accounts/tests/test_views.py (ch21l020)

```
@@ -63,7 +63,7 @@ class LoginViewTest(TestCase):
         response = self.client.get("/accounts/login?token=abcd123")
         self.assertRedirects(response, "/")

-    def DONT_test_logs_in_if_given_valid_token(self):
+    def test_logs_in_if_given_valid_token(self):
         anon_user = auth.get_user(self.client)
         self.assertEqual(anon_user.is_authenticated, False)

@@ -74,7 +74,7 @@ class LoginViewTest(TestCase):
         self.assertEqual(user.is_authenticated, True)
         self.assertEqual(user.email, "edith@example.com")

-    def DONT_test_shows_login_error_if_token_invalid(self):
+    def test_shows_login_error_if_token_invalid(self):
         response = self.client.get("/accounts/login?token=invalid-token", follow=True)
```

Sure enough, they both pass:

```
$ python src/manage.py test accounts
[...]
Ran 20 tests in 0.025s

OK
```

Deciding Which Tests to Keep

We now definitely have duplicate tests:

src/accounts/tests/test_views.py

```
class LoginViewTest(TestCase):
    def test_redirects_to_home_page(self):
        [...]

    def test_logs_in_if_given_valid_token(self):
        [...]

    def test_shows_login_error_if_token_invalid(self):
        [...]

    @mock.patch("accounts.views.auth")
    def test_calls_authenticate_with_uid_from_get_request(self, mock_auth):
        [...]

    @mock.patch("accounts.views.auth")
    def test_calls_auth_login_with_user_if_there_is_one(self, mock_auth):
        [...]

    @mock.patch("accounts.views.auth")
    def test_adds_error_message_if_auth_user_is_None(self, mock_auth):
        [...]
```

The redirect test could stay the same whether we're using mocks or not. We then have two non-mocky tests for the happy and unhappy paths, and three mocky tests:

1. One checks that we are integrated with our auth backend correctly.
2. One checks that we call the built-in `auth.login` function correctly, which tests the happy path.
3. And one checks that we set an error message in the unhappy path.

I think there are lots of ways to justify different choices here, but my instinct tends to be to avoid using mocks if you can. So, I propose we delete the two mocky tests for the happy and unhappy paths, as they are reasonably covered by the non-mocky ones. But I think we can justify keeping the first mocky test, because it adds value by checking that we're doing our authentication the "right" way—i.e., by calling into Django's `auth.authenticate()` function (instead of, for example, instantiating and calling our auth backend ourselves, or even just implementing authentication inline in the view).

"Test behaviour, not implementation" is a GREAT rule of thumb for tests. But sometimes, the fact that you're using one implementation rather than another really is important. In these cases, a mocky test can be useful.

So let's delete our last two mocky tests. I'm also going to rename the remaining one to make our intention clear; we want to check we are using the Django auth library:

src/accounts/tests/test_views.py (ch21l021)

```
@mock.patch("accounts.views.auth")
def test_calls_django_auth_authenticate(self, mock_auth):
    [...]
```

And we're down to 17 tests:

```
$ python src/manage.py test accounts
[...]
Ran 18 tests in 0.015s

OK
```

The Moment of Truth: Will the FT Pass?

We're just about ready to try our functional test! Let's just make sure our base template shows a different navbar for logged-in and non–logged-in users. Our FT relies on being able to see the user's email in the navbar in the logged-in state, and it needs a "Log out" button too:

src/lists/templates/base.html (ch21l022)

```
<nav class="navbar">
  <div class="container-fluid">
    <a class="navbar-brand" href="/">Superlists</a>
    {% if user.email %}  ❶
      <span class="navbar-text">Logged in as {{ user.email }}</span>
      <form method="POST" action="TODO">
        {% csrf_token %}
        <button id="id_logout" class="btn btn-outline-secondary" type="submit">
          Log out
        </button>
      </form>
    {% else %}
      <form method="POST" action="{% url 'send_login_email' %}">
        <div class="input-group">
          <label class="navbar-text me-2" for="id_email_input">
            Enter your email to log in
          </label>
          <input
            id="id_email_input"
            name="email"
            class="form-control"
            placeholder="your@email.com"
          />
          {% csrf_token %}
        </div>
      </form>
    {% endif %}
  </div>
</nav>
```

❶ Here's a new `{% if %}`, and navbar content for logged-in users.

OK, there's a to-do in there about the log-out button. We'll get to that, but how does our FT look now?

```
$ python src/manage.py test functional_tests.test_login
[...]
.
 ----------------------------------------------------------------------
Ran 1 test in 3.282s

OK
```

It Works in Theory! Does It Work in Practice?

Wow! Can you believe it? I scarcely can! Time for a manual look around with `runserver`:

```
$ python src/manage.py runserver
[...]
Internal Server Error: /accounts/send_login_email
Traceback (most recent call last):
  File "...goat-book/accounts/views.py", line 20, in send_login_email

ConnectionRefusedError: [Errno 111] Connection refused
```

Using Our New Environment Variable, and Saving It to .env

You'll probably get an error, like I did, when you try to run things manually. It's because of two things.

Firstly, we need to re-add the email configuration to *settings.py*:

src/superlists/settings.py (ch21l023)

```
EMAIL_HOST = "smtp.gmail.com"
EMAIL_HOST_USER = "obeythetestinggoat@gmail.com"
EMAIL_HOST_PASSWORD = os.environ.get("EMAIL_PASSWORD")
EMAIL_PORT = 587
EMAIL_USE_TLS = True
```

Secondly, we (probably) need to reset the `EMAIL_PASSWORD` in our shell:

```
$ export EMAIL_PASSWORD="yoursekritpasswordhere"
```

Using a Local .env File for Development

Until now, we've not needed to "save" any of our local environment variables, because the command-line ones are easy to remember and type, and we've made sure all the other ones that affect config settings have sensible defaults for dev. But there's just no way to get a working login system without this one!

Rather than having to go look up this password every time you start a new shell, it's quite common to save these sorts of settings into a local file in your project folder named `.env`. It's a convention that makes it a hidden file, on Unix-like systems at least:

```
$ echo .env >> .gitignore  # we don't want to commit our secrets into git!
$ echo EMAIL_PASSWORD="yoursekritpasswordhere" >> .env
$ set -a; source .env; set +a;
```

It does mean you have to remember to do that weird `set -a; source...` dance, every time you start working on the project, as well as remembering to activate your virtualenv.

If you search or ask around, you'll find there are some tools and shell plugins that load virtualenvs and *.env* files automatically, or Django plugins that handle this stuff too. A few options:

- Django-specific: django-environ or django-dotenv
- More general Python project management: Pipenv
- Or even: roll your own

And now…

```
$ python src/manage.py runserver
```

…you should see something like Figure 21-2.

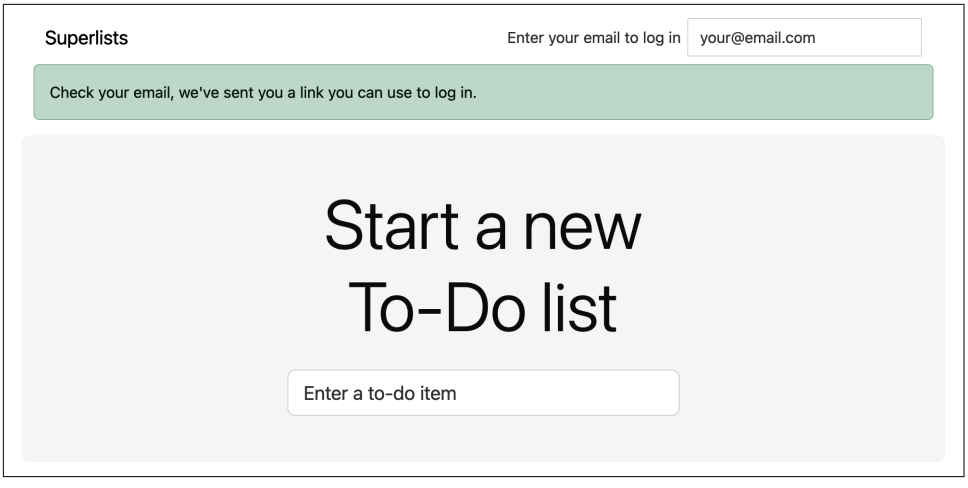
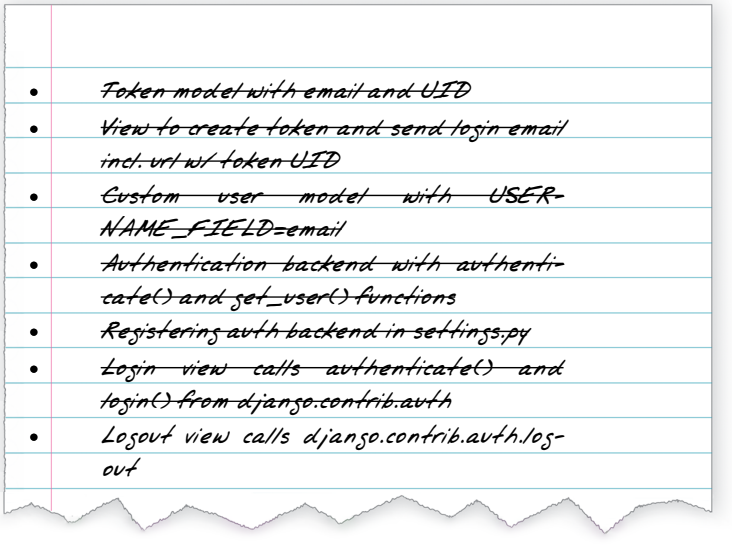


Figure 21-2. Check your email…

Woohoo!

I've been waiting to do a commit up until this moment, just to make sure everything works. At this point, you could make a series of separate commits—one for the login view, one for the auth backend, one for the user model, one for wiring up the template. Or you could decide that—because they're all interrelated, and none will work without the others—you may as well just have one big commit:

```
$ git status
$ git add .
$ git diff --staged
$ git commit -m "Custom passwordless auth backend + custom user model"
```



Finishing Off Our FT: Testing Logout

The last thing we need to do before we call it a day is to test the logout button. We extend the FT with a couple more steps:

src/functional_tests/test_login.py (ch21l024)

```
[...]
# she is logged in!
self.wait_for(
    lambda: self.browser.find_element(By.CSS_SELECTOR, "#id_logout"),
)
navbar = self.browser.find_element(By.CSS_SELECTOR, ".navbar")
self.assertIn(TEST_EMAIL, navbar.text)

# Now she logs out
self.browser.find_element(By.CSS_SELECTOR, "#id_logout").click()

# She is logged out
self.wait_for(
    lambda: self.browser.find_element(By.CSS_SELECTOR, "input[name=email]")
)
navbar = self.browser.find_element(By.CSS_SELECTOR, ".navbar")
self.assertNotIn(TEST_EMAIL, navbar.text)
```

With that, we can see that the test is failing because the logout button doesn't have a valid URL to submit to:

```
$ python src/manage.py test functional_tests.test_login
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: input[name=email]; [...]
```

So, let's tell the base template that we want a new URL named "logout":

src/lists/templates/base.html (ch21l025)

```
{% if user.email %}
  <span class="navbar-text">Logged in as {{ user.email }}</span>
  <form method="POST" action="{% url 'logout' %}">
    {% csrf_token %}
    <button id="id_logout" class="btn btn-outline-secondary" type="submit">
      Log out
    </button>
  </form>
{% else %}
```

If you try the FTs at this point, you'll see an error saying that the URL doesn't exist yet:

```
$ python src/manage.py test functional_tests.test_login
Internal Server Error: /
[...]
django.urls.exceptions.NoReverseMatch: Reverse for 'logout' not found. 'logout'
is not a valid view function or pattern name.

======================================================================
ERROR: test_login_using_magic_link
(functional_tests.test_login.LoginTest.test_login_using_magic_link)
[...]

selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: #id_logout; [...]
```

Implementing a logout URL is actually very simple: we can use Django's built-in logout view, which clears down the user's session and redirects them to a page of our choice:

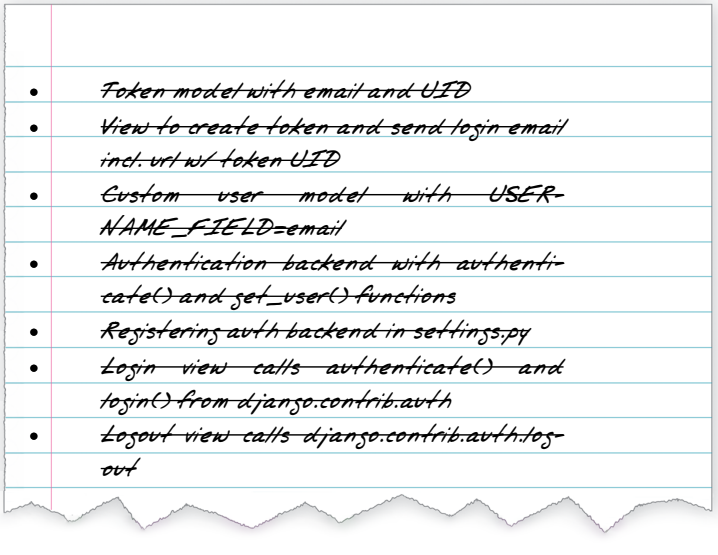
src/accounts/urls.py (ch21l026)

```
from django.contrib.auth import views as auth_views
from django.urls import path

from . import views

urlpatterns = [
    path("send_login_email", views.send_login_email, name="send_login_email"),
    path("login", views.login, name="login"),
    path("logout", auth_views.LogoutView.as_view(next_page="/"), name="logout"),
]
```

And that gets us a fully passing FT—indeed, a fully passing test suite:

```
$ python src/manage.py test functional_tests.test_login
[...]
OK
$ cd src && python manage.py test
[...]
Ran 56 tests in 78.124s

OK
```

We're nowhere near a truly secure or acceptable login system here. As this is just an example app for a book, we'll leave it at that, but in "real life" you'd want to explore a lot more security and usability issues before calling the job done. We're dangerously close to "rolling our own crypto" here, and relying on a more established login system would be much safer. Read more at *https://security.stackexchange.com/a/18198*.



In the next chapter, we'll start trying to put our login system to good use. In the meantime, do a commit and enjoy this recap.

On Mocking in Python

Using mock.return_value
: The `.return_value` attribute on a mock can be used in two ways. You can *read* it, to access the return value of a mocked-out function, and thus check on how it gets used later in your code; this usually happens in the "assert" or "then" part of your test. Alternatively, you can *assign* to it, usally up-front in the "arrange" or "given" part of your test, as a way of saying "I want this mocked-out function to return a particular value".

Mocks can ensure test isolation and reduce duplication
: You can use mocks to isolate different parts of your code from each other, and thus test them independently. This can help you to avoid duplication, because you're only testing a single layer at a time, rather than having to think about combinations of interactions of different layers. Used extensively, this approach leads to "London-style" TDD, but that's quite different from the style I mostly follow and show in this book.

Mocks can enable you to verify implementation details
: Most tests should test behaviour, not implementation. At some point though, we decided using a particular implementation *was* important. And so, we used a mock as a way to verify that, and to document it for our future selves.

There are alternatives to mocks, but they require rethinking how your code is structured
: In a way, mocks make it "too easy". In programming languages that lack Python's dynamic ability to monkeypatch things at runtime, developers have had to work on alternative ways to test code with dependencies. While these techniques can be more complex, they do force you to think about how your code is structured—to cleanly identify your dependencies and build clean abstractions and interfaces around them. Further discussion is beyond the scope of this book, but check out Cosmic Python.

CHAPTER 22

Test Fixtures and a Decorator for Explicit Waits

Now that we have a functional authentication system, we want to use it to identify users, and to show them all the lists they have created.

To do that, we're going to have to write FTs that have a logged-in user. Rather than making each test go through the (time-consuming) login email dance, we want to be able to skip that part.

This is about separation of concerns. Functional tests aren't like unit tests, in that they don't usually have a single assertion. But, conceptually, they should be testing a single thing. There's no need for every single FT to test the login/logout mechanisms. If we can figure out a way to "cheat" and skip that part, we'll spend less time waiting for tests to repeat these duplicated setup steps.

Don't overdo de-duplication in FTs. One of the benefits of an FT is that it can catch strange and unpredictable interactions between different parts of your application.

In this short chapter, we'll start writing our new FT, and use that as an opportunity to talk about de-duplication using test fixtures for FTs. We'll also refactor out a nice helper for explicit waits, using Python's lovely decorator syntax.

Skipping the Login Process by Pre-creating a Session

It's quite common for a user to return to a site and still have a cookie, which means they are "pre-authenticated", so this isn't an unrealistic cheat at all. Here's how you can set it up:

src/functional_tests/test_my_lists.py (ch22l001)

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
from django.contrib.sessions.backends.db import SessionStore

from .base import FunctionalTest

User = get_user_model()


class MyListsTest(FunctionalTest):
    def create_pre_authenticated_session(self, email):
        user = User.objects.create(email=email)
        session = SessionStore()
        session[SESSION_KEY] = user.pk ❶
        session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
        session.save()
        ## to set a cookie we need to first visit the domain.
        ## 404 pages load the quickest!
        self.browser.get(self.live_server_url + "/404_no_such_url/")
        self.browser.add_cookie(
            dict(
                name=settings.SESSION_COOKIE_NAME,
                value=session.session_key,  ❷
                path="/",
            )
        )
```

❶ We create a session object in the database. The session key is the primary key of the user object (which is actually the user's email address).

❷ We then add a cookie to the browser that matches the session on the server—on our next visit to the site, the server should recognise us as a logged-in user.

Note that, as it is, this will only work because we're using `LiveServerTestCase`, so the `User` and `Session` objects we create will end up in the same database as the test server. At some point, we'll need to think about how this will work against Docker or staging.

Django Sessions: How a User's Cookies Tell the Server They Are Authenticated

This is an attempt to explain sessions, cookies, and authentication in Django.

HTTP is a "stateless" protocol, meaning that the protocol itself doesn't keep track of any state from one request to the next, and each request is independent of the next. There's no built-in way to tell that a series of requests come from the same client.

For this reason, servers need a way of recognising different clients with *every single request*. The usual solution is to give each client a unique session ID, which the browser will store in a text file called a "cookie" and send with every request.

The server will store that ID somewhere (by default, in the database), and then it can recognise each request that comes in as being from a particular client.

If you log in to the site using the dev server, you can actually take a look at your session ID by hand if you like. It's stored under the key `sessionid` by default. See Figure 22-1.

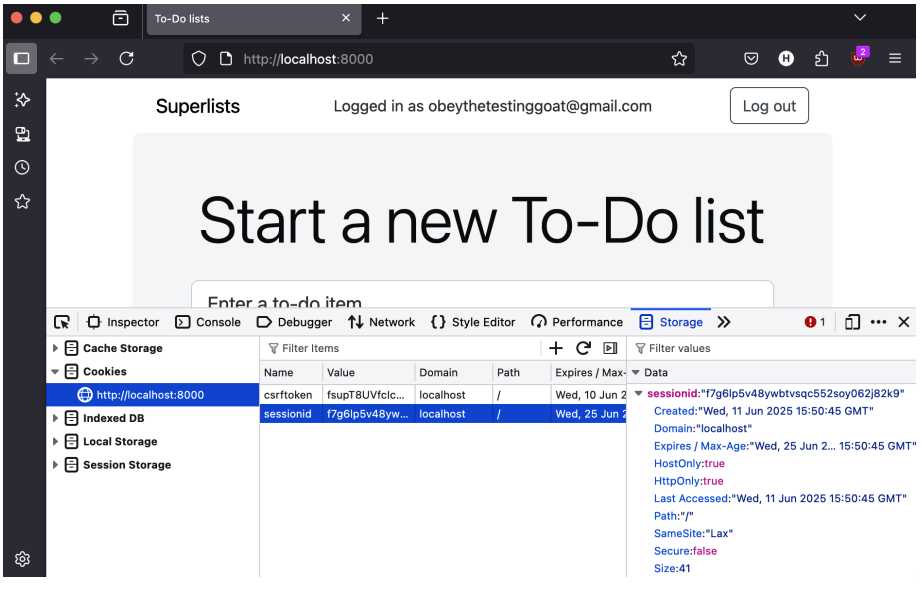


Figure 22-1. Examining the session cookie in the DevTools UI

These session cookies are set for all visitors to a Django site, whether they're logged in or not.

When we want to recognise a client as being a logged-in and authenticated user, again, rather than asking the client to send their username and password with every

single request, the server can actually just mark that client's session as authenticated, and associate it with a user ID in its database.

A Django session is a dictionary-like data structure, and the user ID is stored under the key given by `django.contrib.auth.SESSION_KEY`. You can check this out in a `./manage.py shell` if you like:

```
$ python src/manage.py shell
[...]
In [1]: from django.contrib.sessions.models import Session

# substitute your session id from your browser cookie here
In [2]: session = Session.objects.get(
    session_key="8u0pygdy9blo696g3n4o078ygt6l8y0y"
)

In [3]: print(session.get_decoded())
{'_auth_user_id': 'obeythetestinggoat@gmail.com', '_auth_user_backend':
'accounts.authentication.PasswordlessAuthenticationBackend'}
```

You can also store any other information you like on a user's session, as a way of temporarily keeping track of some state. This works for non–logged-in users too. Just use `request.session` inside any view, and it works as a dictionary. There's more information in the Django docs on sessions.

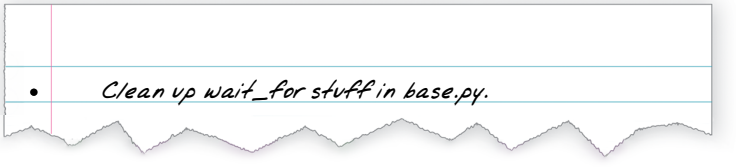
Checking That It Works

To check that the `create_pre_authenticated_session()` system works, it would be good to reuse some of the code from our previous test. Let's make a couple of functions: `wait_to_be_logged_in` and `wait_to_be_logged_out`. To access them from a different test, we'll need to pull them up into `FunctionalTest`. We'll also tweak them slightly so that they can take an arbitrary email address as a parameter:

src/functional_tests/base.py (ch22l002)

```
class FunctionalTest(StaticLiveServerTestCase):
    [...]

    def wait_to_be_logged_in(self, email):
        self.wait_for(
            lambda: self.browser.find_element(By.CSS_SELECTOR, "#id_logout"),
        )
        navbar = self.browser.find_element(By.CSS_SELECTOR, ".navbar")
        self.assertIn(email, navbar.text)

    def wait_to_be_logged_out(self, email):
        self.wait_for(
            lambda: self.browser.find_element(By.CSS_SELECTOR, "input[name=email]")
        )
        navbar = self.browser.find_element(By.CSS_SELECTOR, ".navbar")
        self.assertNotIn(email, navbar.text)
```

Hmm, that's not bad. But I'm not quite happy with the amount of duplication of `wait_for` stuff in here. Let's make a note to come back to it and let's first get these helpers working:



First, we use them in *test_login.py*:

src/functional_tests/test_login.py (ch22l003)

```
def test_login_using_magic_link(self):
    [...]
    # she is logged in!
    self.wait_to_be_logged_in(email=TEST_EMAIL)

    # Now she logs out
    self.browser.find_element(By.CSS_SELECTOR, "#id_logout").click()

    # She is logged out
    self.wait_to_be_logged_out(email=TEST_EMAIL)
```

Just to make sure we haven't broken anything, we rerun the login test:

```
$ python src/manage.py test functional_tests.test_login
[...]
OK
```

And now we can write a placeholder for the "My lists" test, to see if our pre-authenticated session creator really does work:

src/functional_tests/test_my_lists.py (ch22l004)

```
def test_logged_in_users_lists_are_saved_as_my_lists(self):
    email = "edith@example.com"
    self.browser.get(self.live_server_url)
    self.wait_to_be_logged_out(email)

    # Edith is a logged-in user
    self.create_pre_authenticated_session(email)
    self.browser.get(self.live_server_url)
    self.wait_to_be_logged_in(email)
```

That gets us:

```
$ python src/manage.py test functional_tests.test_my_lists
[...]
OK
```

That's a good place for a commit:

```
$ git add src/functional_tests
$ git commit -m "test_my_lists: precreate sessions, move login checks into base"
```

JSON Test Fixtures Considered Harmful

When we pre-populate the database with test data—as we've done here with the `User` object and its associated `Session` object—what we're doing is setting up what's called a "test fixture".

If you look up "Django fixtures", you'll find that Django has a built-in way of saving objects from your database using JSON (using `manage.py dumpdata`), and automatically loading them in your test runs using the `fixtures` class attribute on `TestCase`.

You'll find people out there saying not to use JSON fixtures, and I tend to agree. They're a nightmare to maintain when your model changes. Plus, it's difficult for the reader to tell which of the many attribute values specified in the JSON are critical for the behaviour under test, and which of them are just filler.

Finally, even if tests start out sharing fixtures, sooner or later one test will want slightly different versions of the data, and you end up copying the whole thing around to keep them isolated. Again, it's hard to tell what's relevant to the test and what is just happenstance.

It's usually much more straightforward to just load the data directly using the Django ORM.

Once you have more than a handful of fields on a model, and/or several related models, you'll want to factor out some nice helper methods with descriptive names to build out your data. A lot of people also like `factory_boy`, but I think the most important thing is the descriptive names.

Our Final Explicit Wait Helper: A Wait Decorator

We've used decorators a few times in our code so far, but it's time to learn how they actually work by making one of our own. First, let's imagine how we might want our decorator to work. It would be nice to be able to replace all the custom wait/retry/timeout logic in `wait_for_row_in_list_table()` and the inline `self.wait_fors()` in the `wait_to_be_logged_in/out`. Something like this would look lovely:

src/functional_tests/base.py (ch22l005)

```
    @wait
    def wait_for_row_in_list_table(self, row_text):
        rows = self.browser.find_elements(By.CSS_SELECTOR, "#id_list_table tr")
        self.assertIn(row_text, [row.text for row in rows])

    @wait
    def wait_to_be_logged_in(self, email):
        self.browser.find_element(By.CSS_SELECTOR, "#id_logout")
        navbar = self.browser.find_element(By.CSS_SELECTOR, ".navbar")
        self.assertIn(email, navbar.text)

    @wait
    def wait_to_be_logged_out(self, email):
        self.browser.find_element(By.CSS_SELECTOR, "input[name=email]")
        navbar = self.browser.find_element(By.CSS_SELECTOR, ".navbar")
        self.assertNotIn(email, navbar.text)
```

Are you ready to dive in? Although decorators are quite difficult to wrap your head around,[1] the nice thing is that we've already dipped our toes into functional programming in our `self.wait_for()` helper function. That's a function that takes another function as an argument—and a decorator is the same. The difference is that the decorator doesn't actually execute any code itself; it returns a modified version of the function that it was given.

Our decorator wants to return a new function, which will keep retrying the function being decorated—catching our usual exceptions until a timeout occurs. Here's a first cut:

src/functional_tests/base.py (ch22l006)

```
def wait(fn):  ❶
    def modified_fn():  ❸
        start_time = time.time()
        while True:  ❹
            try:
                return fn()  ❺
            except (AssertionError, WebDriverException) as e:  ❹
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)

    return modified_fn  ❷
```

1 I know it took me a long time before I was comfortable with them, and I still have to think about them quite carefully whenever I make one.

❶ A decorator is a way of modifying a function; it takes a function as an argument…

❷ …and returns another function as the modified (or "decorated") version.

❸ Here's where we define our modified function.

❹ And here's our familiar loop, which will keep catching those exceptions until the timeout.

❺ And as always, we call our original function and return immediately if there are no exceptions.

That's *almost* right, but not quite; try running it?

```
$ python src/manage.py test functional_tests.test_my_lists
[...]
    self.wait_to_be_logged_out(email)
TypeError: wait.<locals>.modified_fn() takes 0 positional arguments but 2 were
given
```

Unlike in `self.wait_for`, the decorator is being applied to functions that have arguments:

src/functional_tests/base.py

```
    @wait
    def wait_to_be_logged_in(self, email):
        self.browser.find_element(By.CSS_SELECTOR, "#id_logout")
        [...]
```

`wait_to_be_logged_in` takes `self` and `email` as positional arguments. But when it's decorated, it's replaced with `modified_fn`, which currently takes no arguments. How do we magically make it so our `modified_fn` can handle the same arguments as whatever function the decorator is given?

The answer is a bit of Python magic, `*args` and `**kwargs`, more formally known as "variadic arguments" (apparently—I only just learned that):

src/functional_tests/base.py (ch22l007)

```
def wait(fn):
    def modified_fn(*args, **kwargs):  ❶
        start_time = time.time()
        while True:
            try:
                return fn(*args, **kwargs)  ❷
            except (AssertionError, WebDriverException) as e:
                if time.time() - start_time > MAX_WAIT:
                    raise e
                time.sleep(0.5)

    return modified_fn
```

❶ Using `*args` and `**kwargs`, we specify that `modified_fn()` may take any arbitrary positional and keyword arguments.

❷ As we've captured them in the function definition, we make sure to pass those same arguments to `fn()` when we actually call it.

One of the fun things this can be used for is to make a decorator that changes the arguments of a function. But we won't get into that now. The main thing is that our decorator now works!

```
$ python src/manage.py test functional_tests.test_my_lists
[...]
OK
```

And do you know what's truly satisfying? We can use our `wait` decorator for our `self.wait_for` helper as well! Like this:

src/functional_tests/base.py (ch22l008)

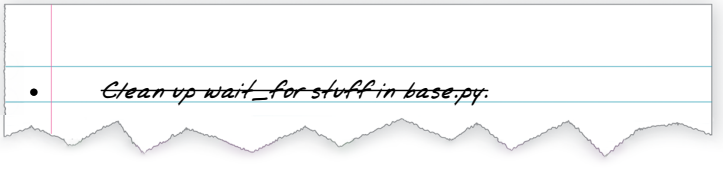
```
    @wait
    def wait_for(self, fn):
        return fn()
```

Lovely! Now all our wait/retry logic is encapsulated in a single place, and we have a nice easy way of applying those waits—either inline in our FTs using `self.wait_for()`, or on any helper function using the `@wait` decorator.

Let's just check all the FTs still pass of course:

```
Ran 8 tests in 19.379s

OK
```

Do a commit, and we're good to cross off that scratchpad item:



In the next chapter, we'll try to deploy our code to staging, and use the pre-authenticated session fixtures on the server. As we'll see, it'll help us catch a little bug or two!

Lessons Learned

Decorators
: Decorators can be a great way of abstracting out different levels of concerns. They let us write our test assertions without having to think about waits at the same time.

De-duplicating your FTs, with caution
: Every single FT doesn't need to test every single part of your application. In our case, we wanted to avoid going through the full login process for every FT that needs an authenticated user, so we used a test fixture to "cheat" and skip that part. You might find other things you want to skip in your FTs. A word of caution, however: functional tests are there to catch unpredictable interactions between different parts of your application, so be wary of pushing de-duplication to the extreme.

Test fixtures
: Test fixtures refers to test data that needs to be set up as a precondition before a test is run—often this means populating the database with some information, but as we've seen (with browser cookies), it can involve other types of preconditions.

Avoiding JSON fixtures
: Django makes it easy to save and restore data from the database in JSON format (and others) using the `dumpdata` and `loaddata` management commands. I would tend to recommend against them, as they are painful to manage when your database schema changes. Use the ORM, with some nicely named helper functions instead.

CHAPTER 23

Debugging and Testing Server Issues

Popping a few layers off the stack of things we're working on: we have nice wait-for helpers; what were we using them for? Oh yes, waiting to be logged in. And why was that? Ah yes, we had just built a way of pre-authenticating a user. Let's see how that works against Docker and our staging server.

The Proof Is in the Pudding: Using Docker to Catch Final Bugs

Remember the deployment checklist from Chapter 18? Let's see if it can't come in handy today!

First, we rebuild and start our Docker container locally, on port 8888:

```
$ docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/container.db.sqlite3",target=/home/nonroot/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -e DJANGO_DB_PATH=/home/nonroot/db.sqlite3 \
    -it superlists
[...]
 => => naming to docker.io/library/superlists [...]
[2025-01-27 22:37:02 +0000] [7] [INFO] Starting gunicorn 22.0.0
[2025-01-27 22:37:02 +0000] [7] [INFO] Listening at: http://0.0.0.0:8888 (7)
[2025-01-27 22:37:02 +0000] [7] [INFO] Using worker: sync
[2025-01-27 22:37:02 +0000] [8] [INFO] Booting worker with pid: 8
```

If you see an error saying `bind source path does not exist`, you've lost your container database somehow. Create a new one with `touch container.db.sqlite3`.

Now let's make sure our container database is fully up to date, by running `migrate` inside the container:

```
$ docker exec $(docker ps --filter=ancestor=superlists -q) python manage.py migrate
Operations to perform:
  Apply all migrations: accounts, auth, contenttypes, lists, sessions
Running migrations:
[...]
```

That little `$(docker ps --filter=ancestor=superlists -q)` is a neat way to avoid manually looking up the container ID. An alternative would be to just set the container name explicitly in our `docker run` commands, using `--name`.

And now, let's do an FT run:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: #id_logout; [...]
[...]
AssertionError: 'Check your email' not found in 'Server Error (500)'
[...]
FAILED (failures=1, errors=1)
```

We can't log in—either with the real email system or with our pre-authenticated session. Looks like our nice new authentication system is crashing when we run it in Docker.

Let's practice a bit of production debugging!

Inspecting the Docker Container Logs

When Django fails with a 500 or "unhandled exception" and `DEBUG` is off, it doesn't print the tracebacks to your web browser. But it will send them to your logs instead.

Check Our Django LOGGING Settings

It's worth double-checking at this point that your *settings.py* still contains the `LOGGING` settings that will actually send stuff to the console:

src/superlists/settings.py

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": False,
    "handlers": {
        "console": {"class": "logging.StreamHandler"},
    },
    "loggers": {
        "root": {"handlers": ["console"], "level": "INFO"},
    },
}
```

Rebuild and restart the Docker container if necessary, and then either rerun the FT, or just try to log in manually.

If you switch to the terminal that's running your Docker image, you should see the traceback printed out in there:

```
Internal Server Error: /accounts/send_login_email
Traceback (most recent call last):
[...]
  File "/src/accounts/views.py", line 16, in send_login_email
    send_mail(
    ~~~~~~~~~^
        "Your login link for Superlists",
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    ...<2 lines>...
        [email],
        ^^^^^^^^
    )
    ^
[...]
    self.connection.sendmail(
    ~~~~~~~~~~~~~~~~~~~~~~~~^
        from_email, recipients, message.as_bytes(linesep="\r\n")
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    )
    ^
  File "/usr/local/lib/python3.14/smtplib.py", line 876, in sendmail
    raise SMTPSenderRefused(code, resp, from_addr)
smtplib.SMTPSenderRefused: (530, b'5.7.0 Authentication Required. [...]
```

Sure enough, that looks like a pretty good clue as to what's going on: we're getting a "sender refused" error when trying to send our email. Good to know our local Docker setup can reproduce the error on the server!

Another Environment Variable in Docker

So, Gmail is refusing to let us send emails, is it? Now why might that be? "Authentication required", you say? Oh, whoops; we haven't told the server what our password is!

As you might remember from earlier chapters, our *settings.py* expects to get the email server password from an environment variable named `EMAIL_PASSWORD`:

src/superlists/settings.py

```
EMAIL_HOST_PASSWORD = os.environ.get("EMAIL_PASSWORD")
```

Let's add this new environment variable to our local Docker container `run` command. First, set your email password in your terminal if you need to:

```
$ echo $EMAIL_PASSWORD
# if that's empty, let's set it:
$ export EMAIL_PASSWORD="yoursekritpasswordhere"
```

Now let's pass that environment variable through to our Docker container using one more `-e` flag—this one fishing the env var out of the shell we're in:

```
$ docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/container.db.sqlite3",target=/home/nonroot/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -e DJANGO_DB_PATH=/home/nonroot/db.sqlite3 \
    -e EMAIL_PASSWORD \
    -it superlists
```

If you use `-e` without the `=something` argument, it sets the env var inside Docker to the same value set in the current shell. It's like saying `-e EMAIL_PASSWORD=$EMAIL_PASSWORD`.

And now we can rerun our FT again. We'll narrow it down to just the `test_login` test, because that's the main one that has a problem:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests.test_login
[...]
ERROR: test_login_using_magic_link
(functional_tests.test_login.LoginTest.test_login_using_magic_link)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/functional_tests/test_login.py", line 32, in
test_login_using_magic_link
    email = mail.outbox.pop()
IndexError: pop from empty list
```

Well, not a pass, but the tests do get a little further. It looks like our server *can* now send emails. (If you check the Docker logs, you'll see there are no more errors.) But our FT is saying it can't see any emails appearing in `mail.outbox`.

mail.outbox Won't Work Outside Django's Test Environment

The reason is that `mail.outbox` is a local, in-memory variable in Django, so that's only going to work when our tests and our server are running in the same process—like they do with unit tests or with `LiveServerTestCase` FTs.

When we run against another process, be it Docker or an actual server, we can't access the same `mail.outbox` variable. If we want to actually inspect the emails that the server sends we need another technique in our tests against Docker (or later, against the staging server).

Deciding How to Test "Real" Email Sending

This is a point at which we have to explore some trade-offs. There are a few different ways we could test email sending:

1. We could build a "real" end-to-end test, and have our tests log in to an email server using the POP3 protocol to retrieve the email from there. That's what I did in the first and second editions of this book.
2. We can use a service like Mailinator or Mailsac, which gives us an email account to send to, along with APIs for checking what mail has been delivered.
3. We can use an alternative, fake email backend whereby Django will save the emails to a file on disk, for example, and we can inspect them there.
4. We could give up on testing email on the server. If we have a minimal smoke test confirming that the server *can* send emails, then we don't need to test that they are actually delivered.

Table 23-1 lays out some of the pros and cons.

Table 23-1. Testing strategy trade-offs

Strategy	Pros	Cons
End-to-end with POP3	Maximally realistic, tests the whole system	Slow, fiddly, unreliable
Email testing service e.g., Mailinator/Mailsac	As realistic as real POP3, with better APIs for testing	Slow, possibly expensive (and I don't want to endorse any particular commercial provider)
File-based fake email backend	Faster, more reliable, no network calls, tests end-to-end (albeit with fake components)	Still fiddly, requires managing database and filesystem side effects
Giving up on testing email on the server/Docker	Fast, simple	Less confidence that things work "for real"

We're exploring a common problem in testing integration with external systems; how far should we go? How realistic should we make our tests?

In this case, I'm going to suggest we go for the last option, which is *not* to test email sending on the server or in Docker. Email itself is a well-understood protocol (reader, it's been around since *before I was born*, and that's a while ago now), and Django has supported sending email for more than a decade. So, I think we can afford to say, in this case, that the costs of building testing tools for email outweigh the benefits.

I'm going to suggest we stick to using `mail.outbox` when we're running local tests, and we configure our FTs to just check that Docker (or, later, the staging server) *seems* to be able to send email (in the sense of "not crashing"). We can skip the bit where we check the email contents in our FT. Remember, we also have unit tests for the email content!

I explore some of the difficulties involved in getting these kinds of tests to work in Online Appendix: Functional Tests for External Dependencies, so go check that out if this feels like a bit of a cop-out!

Here's where we can put an early return in the FT:

src/functional_tests/test_login.py (ch23l009)

```
        # A message appears telling her an email has been sent
        self.wait_for(
            lambda: self.assertIn(
                "Check your email",
                self.browser.find_element(By.CSS_SELECTOR, "body").text,
            )
        )

        if self.test_server:
            # Testing real email sending from the server is not worth it.
            return

        # She checks her email and finds a message
        email = mail.outbox.pop()
```

This test will still fail if you don't set `EMAIL_PASSWORD` to a valid value in Docker or on the server, meaning it would still have warned us of the bug we started the chapter with—so that's good enough for now.

Here's how we populate the `FunctionalTest.test_server` attribute:

src/functional_tests/base.py (ch23l010)

```
class FunctionalTest(StaticLiveServerTestCase):
    def setUp(self):
        self.browser = webdriver.Firefox()
        self.test_server = os.environ.get("TEST_SERVER")  ❶
        if self.test_server:
            self.live_server_url = "http://" + self.test_server
```

❶ We upgrade `test_server` to be an attribute on the test object, so we can access it in various places in our FTs (we'll see several examples later). Sad to see our walrus go, though!

And you can confirm that the FT fails if you *don't* set `EMAIL_PASSWORD` in Docker, or passes, if you do:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests.test_login
[...]

OK
```

Now let's see if we can get our FTs to pass against the server. First, we'll need to figure out how to set that env var on the server.

An Alternative Method for Setting Secret Environment Variables on the Server

In Chapter 12, we dealt with setting the `SECRET_KEY` by generating a random value, and then saving it to a file on the server. We could use a similar technique here. But, just to give you an alternative, I'll show how to pass the environment variable directly up to the container, without storing it in a file:

infra/deploy-playbook.yaml (ch23l012)

```
env:
  DJANGO_DEBUG_FALSE: "1"
  DJANGO_SECRET_KEY: "{{ secret_key.content | b64decode }}"
  DJANGO_ALLOWED_HOST: "{{ inventory_hostname }}"
  DJANGO_DB_PATH: "/home/nonroot/db.sqlite3"
  EMAIL_PASSWORD: "{{ lookup('env', 'EMAIL_PASSWORD') }}"  ❶
```

❶ `lookup()` with `env` as its argument is how you look up *local* environment variables—i.e., the ones set on the computer you're running Ansible from.

This means you'll need the `EMAIL_PASSWORD` environment variable to be set on your local machine every time you want to run Ansible.

Let's consider some pros and cons of the two approaches:

- Saving the secret to a file on the server means you don't need to "remember" or store the secret anywhere on your own machine.
- In contrast, always passing it up from the local environment does mean you can change the value of the secret at any time.
- In terms of security, they are fairly equivalent—in either case, the environment variable is visible via `docker inspect`.

If we rerun our full FT suite against the server, you should see that the login test passes, and we're down to just one failure, in `test_logged_in_users_lists_are_saved_as_my_lists()`:

```
$ TEST_SERVER=staging.ottg.co.uk python src/manage.py test functional_tests
[...]
ERROR: test_logged_in_users_lists_are_saved_as_my_lists
(functional_tests.test_my_lists.MyListsTest.test_logged_in_users_lists_are_saved_[...]
----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/src/functional_tests/test_my_lists.py", line 36, in
test_logged_in_users_lists_are_saved_as_my_lists
    self.wait_to_be_logged_in(email)
    ~~~~~~~~~~~~~~~~~~~~~~~~^^^^^^^
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: #id_logout; [...]
[...]
 ----------------------------------------------------------------------

Ran 8 tests in 30.087s

FAILED (errors=1)
```

Let's look into that next.

Debugging with SQL

Let's switch back to testing locally against our Docker container:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: #id_logout; [...]
FAILED (errors=1)
```

It looks like the attempt to create pre-authenticated sessions doesn't work, so we're not being logged in. Let's do a bit of debugging with SQL.

First, try logging in to your local "runserver" instance (where things definitely work) and take a look in the normal local database, *src/db.sqlite3*:

```
$ sqlite3 src/db.sqlite3
SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.

sqlite> select * from accounts_token;  ❶
1|obeythetestinggoat@gmail.com|11d3e26d-32a3-4434-af71-5e0f62fefc52
2|obeythetestinggoat@gmail.com|25a570c8-736f-42e4-931b-ed5c410b5b51

sqlite> select * from django_session;  ❷
tv2m5byccfs05gfpkc1l8k4pep097y3c|.eJxVjEsKg0AMQO-StcwBurI9gTcYYgwzo[...]
```

❶ We can do a `SELECT *` in our tokens table to see some of the tokens we've been creating for our users.

❷ And we can take a look in the `django_session` table. You should find the first column matches the session ID you'll see in your DevTools.

Let's do a bit of debugging. Take a look in *container.db.sqlite3*:

```
$ sqlite3 container.db.sqlite3
SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.

sqlite> select * from accounts_token;  ❶

sqlite> select * from django_session;  ❷
```

❶ The users table is empty. (If you do see `edith@example.com` in here, it's from a previous test run. Delete and re-create the database if you want to be sure.)

❷ And the sessions table is definitely empty.

Now, let's try manually. If you visit `localhost:8888` and log in—getting the token from your email—you'll see it works. You can also run `functional_tests.test_login` and you'll see *that* pass.

If we look in the database again, we'll see some more data:

```
$ sqlite3 container.db.sqlite3
SQLite version 3.43.2 2023-10-10 13:08:14
Enter ".help" for usage hints.

sqlite> select * from accounts_token;
3|obeythetestinggoat@gmail.com|115812a3-7d37-485c-9c15-337b12293f69
4|edith@example.com|a901bee9-88aa-4965-9277-a13723a6bfe1

sqlite> select * from django_session;
09df51nmvpi137mpv5bwjoghh2a4y5lh|.eJxVjEsKg0AMQO-[...]
```

So, there's nothing *fundamentally* wrong with the Docker environment. It's seems like it's specifically our test utility function `create_pre_authenticated_session()` that isn't working.

At this point, a little niggle in your head might be growing louder, reminding us of a problem we anticipated in the last chapter: `LiveServerTestCase` only lets us talk to the in-memory database. That's where our pre-authenticated sessions are ending up!

Managing Fixtures in Real Databases

We need a way to make changes to the database inside Docker or on the server. Essentially, we want to run some code outside the context of the tests (and the test database) and in the context of the server and its database.

A Django Management Command to Create Sessions

When trying to build a standalone script that works with Django (i.e., can talk to the database and so on), there are some fiddly issues you need to get right, like setting the `DJANGO_SETTINGS_MODULE` environment variable and setting `sys.path` correctly.

Instead of messing about with all that, Django lets you create your own "management commands" (commands you can run with `python manage.py`), which will do all that path-mangling for you. They live in a folder called *management/commands* inside your apps:

```
$ mkdir -p src/functional_tests/management/commands
$ touch src/functional_tests/management/__init__.py
$ touch src/functional_tests/management/commands/__init__.py
```

The boilerplate in a management command is a class that inherits from `django.core.management.BaseCommand`, and that defines a method called `handle`:

src/functional_tests/management/commands/create_session.py (ch23l014)

```
from django.conf import settings
from django.contrib.auth import BACKEND_SESSION_KEY, SESSION_KEY, get_user_model
from django.contrib.sessions.backends.db import SessionStore
from django.core.management.base import BaseCommand

User = get_user_model()


class Command(BaseCommand):
    def add_arguments(self, parser):
        parser.add_argument("email")

    def handle(self, *args, **options):
        session_key = create_pre_authenticated_session(options["email"])
        self.stdout.write(session_key)


def create_pre_authenticated_session(email):
    user = User.objects.create(email=email)
    session = SessionStore()
    session[SESSION_KEY] = user.pk
    session[BACKEND_SESSION_KEY] = settings.AUTHENTICATION_BACKENDS[0]
    session.save()
    return session.session_key
```

We've taken the code for `create_pre_authenticated_session` from *test_my_lists.py*. `handle` will pick up an email address from the parser, and then return the session key that we'll want to add to our browser cookies, and the management command prints it out at the command line.

Try it out:

```
$ python src/manage.py create_session a@b.com
Unknown command: 'create_session'. Did you mean clearsessions?
```

One more step: we need to add `functional_tests` to our *settings.py* so that it's recognised as a real app that might have management commands as well as tests:

src/superlists/settings.py (ch23l015)

```
+++ b/superlists/settings.py
@@ -42,6 +42,7 @@ INSTALLED_APPS = [
     "accounts",
     "lists",
+    "functional_tests",
 ]
```

Beware of the security implications here. We're now adding some remotely executable code for bypassing authentication to our default configuration. Yes, someone exploiting this would need to have already gained access to the server, so it was game over anyway, but nonetheless, this is a sensitive area. If you were doing something like this in a real application, you might consider adding an `if environment != prod`, or similar.

Now it works:

```
$ python src/manage.py create_session a@b.com
qnslckvp2aga7tm6xuivyb0ob1akzzwl
```

If you see an error saying the `auth_user` table is missing, you may need to run `manage.py migrate`. In case that doesn't work, delete the *db.sqlite3* file and run `migrate` again to get a clean slate.

Getting the FT to Run the Management Command on the Server

Next, we need to adjust `test_my_lists` so that it runs the local function when we're using the local in-memory test server from `LiveServerTestCase`. And, if we're running against the Docker container or staging server, it should run the management command instead.

src/functional_tests/test_my_lists.py (ch23l016)

```
from django.conf import settings

from .base import FunctionalTest
from .container_commands import create_session_on_server  ❶
from .management.commands.create_session import create_pre_authenticated_session


class MyListsTest(FunctionalTest):
    def create_pre_authenticated_session(self, email):
        if self.test_server:  ❷
            session_key = create_session_on_server(self.test_server, email)
        else:
            session_key = create_pre_authenticated_session(email)

        ## to set a cookie we need to first visit the domain.
        ## 404 pages load the quickest!
        self.browser.get(self.live_server_url + "/404_no_such_url/")
        self.browser.add_cookie(
            dict(
                name=settings.SESSION_COOKIE_NAME,
                value=session_key,
                path="/",
            )
        )

    [...]
```

❶ Programming by wishful thinking, let's imagine we'll have a module called `container_commands` with a function called `create_session_on_server()` in it.

❷ Here's the `if` where we decide which of our two session-creation functions to execute.

Running Commands Using Docker Exec and (Optionally) SSH

You may remember `docker exec` from Chapter 9; it lets us run commands inside a running Docker container. That's fine for when we're running against the local Docker, but when we're against the server, we need to SSH in first.

There's a bit of plumbing here, but I've tried to break things down into small chunks:

src/functional_tests/container_commands.py (ch23l018)

```
import subprocess

USER = "elspeth"


def create_session_on_server(host, email):
    return _exec_in_container(
        host, ["/venv/bin/python", "/src/manage.py", "create_session", email]  ❶
    )


def _exec_in_container(host, commands):
    if "localhost" in host:  ❷
        return _exec_in_container_locally(commands)
    else:
        return _exec_in_container_on_server(host, commands)


def _exec_in_container_locally(commands):
    print(f"Running {commands} on inside local docker container")
    return _run_commands(["docker", "exec", _get_container_id()] + commands)  ❸


def _exec_in_container_on_server(host, commands):
    print(f"Running {commands!r} on {host} inside docker container")
    return _run_commands(
        ["ssh", f"{USER}@{host}", "docker", "exec", "superlists"] + commands  ❹
    )


def _get_container_id():
    return subprocess.check_output(  ❺
        ["docker", "ps", "-q", "--filter", "ancestor=superlists"]  ❸
    ).strip()


def _run_commands(commands):
    process = subprocess.run(  ❺
        commands,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        check=False,
    )
    result = process.stdout.decode()
    if process.returncode != 0:
        raise Exception(result)
    print(f"Result: {result!r}")
    return result.strip()
```

❶ We invoke our management command with the path to the virtualenv Python, the `create_session` command name, and pass in the email we want to create a session for.

❷ We dispatch to two slightly different ways of running a command inside a container, with the assumption that a host on "localhost" is a local Docker container, and the others are on the staging server.

❸ To run a command on the local Docker container, we're going to use `docker exec`, and we have a little extra hop first to get the correct container ID.

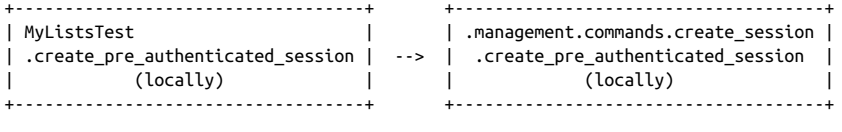
❹ To run a command on the Docker container that's on the staging server, we still use `docker exec`, but we do it inside an SSH session. In this case we don't need the container ID, because the container is always named "superlists".

❺ Finally, we use Python's `subprocess` module to actually run a command. You can see a couple of different ways of running it here, which differ based on how we're handing errors and output; the details don't matter too much.

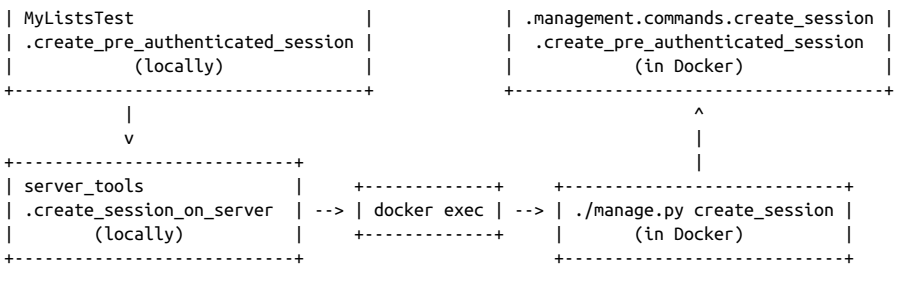
Recap: Creating Sessions Locally Versus Staging

Does that all make sense? Perhaps a little ASCII-art diagram will help:

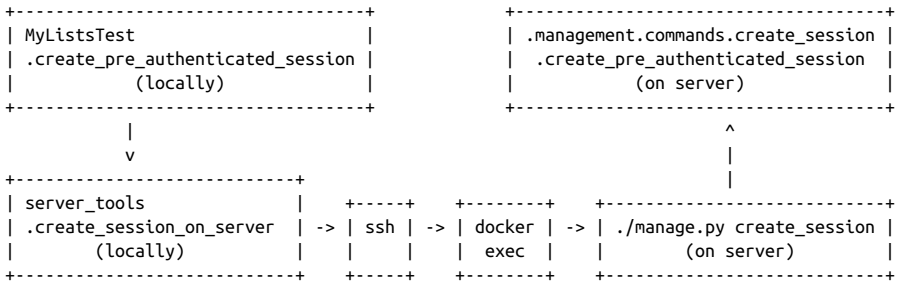
Locally:



Against Docker locally:



Against Docker on the server:



We do love a bit of ASCII art now and again!

An Alternative for Managing Test Database Content: Talking Directly to the Database

An alternative way of managing database content inside Docker, or on a server, would be to talk directly to the database.

Because we're using SQLite, that involves writing to the file directly. This can be fiddly to get right, because when we're running inside Django's test runner, Django takes over the test database creation, so you end up having to write raw SQL and manage your connections to the database directly.

There are also some tricky interactions with the filesystem mounts and Docker, as well as the need to have the `SECRET_KEY` env var set to the same value as on the server.

If we were using a "classic" database server like PostgreSQL or MySQL, we'd be able to talk directly to the database over its port, and that's an approach I've used successfully in the past but it's still quite tricky, and usually requires writing your own SQL.

Testing the Management Command

In any case, let's see if this whole rickety pipeline works. First, locally, to check that we didn't break anything:

```
$ python src/manage.py test functional_tests.test_my_lists
[...]
OK
```

Next, against Docker—rebuild first:

```
$ docker build -t superlists . && docker run \
    -p 8888:8888 \
    --mount type=bind,source="$PWD/container.db.sqlite3",target=/home/nonroot/db.sqlite3 \
    -e DJANGO_SECRET_KEY=sekrit \
    -e DJANGO_ALLOWED_HOST=localhost \
    -e DJANGO_DB_PATH=/home/nonroot/db.sqlite3 \
    -e EMAIL_PASSWORD \
    -it superlists
```

And then we run the FT (that uses our fixture) against Docker:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests.test_my_lists

[...]
OK
```

Next, we run it against the server. First, we re-deploy to make sure our code on the server is up to date:

```
$ ansible-playbook --user=elspeth -i staging.ottg.co.uk, infra/deploy-playbook.yaml -vv
```

And now we run the test:

```
$ TEST_SERVER=staging.ottg.co.uk python src/manage.py test \
 functional_tests.test_my_lists
Found 1 test(s).
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Running '/venv/bin/python /src/manage.py create_session edith@example.com' on
staging.ottg.co.uk inside docker container
Result: '7n032ogf179t2e7z3olv9ct7b3d4dmas\n'
.
 ---------------------------------------------------------------------
Ran 1 test in 4.515s

OK
Destroying test database for alias 'default'...
```

Looking good! We can rerun all the tests to make sure…

```
$ TEST_SERVER=staging.ottg.co.uk python src/manage.py test functional_tests
[...]
[elspeth@staging.ottg.co.uk] run:
~/sites/staging.ottg.co.uk/.venv/bin/python
[...]
Ran 8 tests in 89.494s

OK
```

Hooray!

Test Database Cleanup

One more thing to be aware of: now that we're running against a real database, we don't get cleanup for free any more. If you try running the tests twice—locally or against Docker—you'll run into this error:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests.test_my_lists
[...]
django.db.utils.IntegrityError: UNIQUE constraint failed: accounts_user.email
```

It's because the user we created the first time we ran the tests is still in the database. When we're running against Django's test database, Django cleans up for us. Let's try and emulate that when we're running against a real database:

src/functional_tests/container_commands.py (ch23l019)

```
def reset_database(host):
    return _exec_in_container(
        host, ["/venv/bin/python", "/src/manage.py", "flush", "--noinput"]
    )
```

And let's add the call to `reset_database()` in our base test `setUp()` method:

src/functional_tests/base.py (ch23l020)

```
from .container_commands import reset_database
[...]

class FunctionalTest(StaticLiveServerTestCase):
    def setUp(self):
        self.browser = webdriver.Firefox()
        self.test_server = os.environ.get("TEST_SERVER")
        if self.test_server:
            self.live_server_url = "http://" + self.test_server
            reset_database(self.test_server)
```

If you try to run your tests again, you'll find they pass happily:

```
$ TEST_SERVER=localhost:8888 python src/manage.py test functional_tests.test_my_lists
[...]

OK
```

Probably a good time for a commit! :)

Warning: Be Careful Not to Run Test Code Against the Production Server!

We're in dangerous territory now that we have code that can directly affect a database on the server. You want to be very, very careful that you don't accidentally blow away your production database by running FTs against the wrong host.

You might consider putting some safeguards in place at this point. You almost definitely want to put staging and production on different servers, for example, and make it so that they use different key pairs for authentication, with different passphrases.

I also mentioned not including the FT management commands in `INSTALLED_APPS` for production environments.

This is similarly dangerous territory to running tests against clones of production data. I could tell you a little story about accidentally sending thousands of duplicate invoices to clients, for example. LFMF! And tread carefully.

Wrap-Up

Actually getting your new code up and running on a server always tends to flush out some last-minute bugs and unexpected issues. We had to do a bit of work to get through them, but we've ended up with several useful things as a result.

We now have a lovely generic `wait` decorator, which will be a nice Pythonic helper for our FTs from now on. We've got some more robust logging configuration. We have test fixtures that work both locally and on the server, and we've come out with a pragmatic approach for testing email integration.

But before we can deploy our actual production site, we'd better actually give the users what they wanted—the next chapter describes how to give them the ability to save their lists on a "My lists" page.

Lessons Learned Catching Bugs in Staging

It's nice to be able to repro things locally.
: The effort we put into adapting our app to use Docker is paying off. We discovered an issue in staging, and were able to reproduce it locally. That gives us the ability to experiment and get feedback much quicker than trying to do experiments on the server itself.

Fixtures also have to work remotely.
: `LiveServerTestCase` makes it easy to interact with the test database using the Django ORM for tests running locally. Interacting with the database inside Docker is not so straightforward. One solution is `docker exec` and Django management commands, as I've shown, but you should explore what works for you—connecting directly to the database over SSH tunnels, for example.

Be very careful when resetting data on your servers.
: A command that can remotely wipe the entire database on one of your servers is a dangerous weapon, and you want to be really, really sure it's never accidentally going to hit your production data.

Logging is critical to debugging issues on the server.
: At the very least, you'll want to be able to see any error messages that are being generated by the server. For thornier bugs, you'll also want to be able to do the occasional "debug print", and see it end up in a file somewhere.

CHAPTER 24

Finishing "My Lists": Outside-In TDD

In this chapter, I'd like to talk about a technique called outside-in TDD. It's pretty much what we've been doing all along. Our "double-loop" TDD process, in which we write the functional test first and then the unit tests, is already a manifestation of outside-in—we design the system from the outside, and build up our code in layers. Now I'll make it explicit, and talk about some of the common issues involved.

The Alternative: Inside-Out

The alternative to "outside-in" is to work "inside-out", which is the way most people intuitively work before they encounter TDD. After coming up with a design, the natural inclination is sometimes to implement it starting with the innermost, lowest-level components first.

For example, when faced with our current problem, providing users with a "My lists" page of saved lists, the temptation is to start at the models layer: we probably want to add an "owner" attribute to the `List` model object, reasoning that an attribute like this is "obviously" going to be required. Once that's in place, we would modify the more peripheral layers of code—such as views and templates—taking advantage of the new attribute, and then finally add URL routing to point to the new view.

It feels comfortable because it means you're never working on a bit of code that is dependent on something that hasn't yet been implemented. Each bit of work on the inside is a solid foundation on which to build the next layer out.

But working inside-out like this also has some weaknesses.

Why Prefer "Outside-In"?

The most obvious problem with inside-out TDD is that it requires us to stray from a TDD workflow. Our functional test's first failure might be due to missing URL routing, but we decide to ignore that and go off adding attributes to our database model objects instead.

We might have ideas in our head about the desired behaviour of our inner layers like database models, and often these ideas will be pretty good—but they are actually just speculation about what's really required, because we haven't yet built the outer layers that will use them.

One problem that can occur is building inner components that are more general or more capable than we actually need, which is a waste of time and an added source of complexity for your project. Another common problem is that you create inner components with an API that is convenient for their own internal design, but which later turns out to be inappropriate for the calls that your outer layers would like to make…worse still, you might end up with inner components which, you later realise, don't actually solve the problem that your outer layers need solved.

In contrast, working outside-in enables you to use each layer to imagine the most convenient API you could want from the layer beneath it. Let's see it in action.

The FT for "My Lists"

As we work through the following functional test, we start with the most outward-facing (presentation layer), through to the view functions (or "controllers"), and lastly the innermost layers, which in this case will be model code. See Figure 24-1.

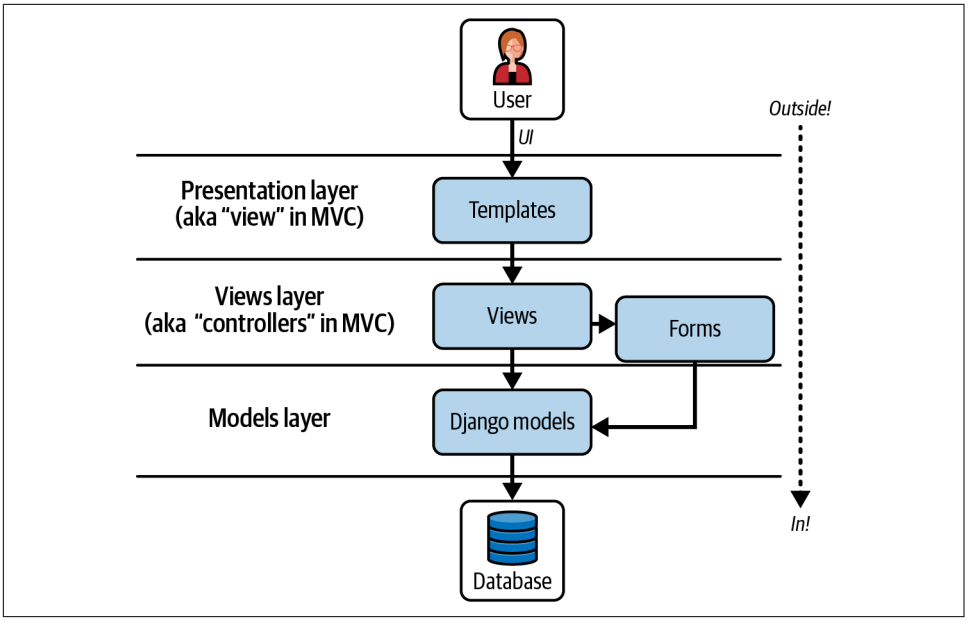


Figure 24-1. The layer in our application

While we're drawing diagrams, would it help to sketch out what we're imagining? See Figure 24-2.

Figure 24-2. A sketch of the "My lists" page

Let's incarnate this idea in FT form. We know our `create_pre_authenticated_session` code works now, so we can just fill out the actual body of the test to describe how a user might interact with this prospective "My lists" page:

src/functional_tests/test_my_lists.py (ch24l001)

```
from selenium.webdriver.common.by import By
[...]

    def test_logged_in_users_lists_are_saved_as_my_lists(self):
        # Edith is a logged-in user
        self.create_pre_authenticated_session("edith@example.com")

        # She goes to the home page and starts a list
        self.browser.get(self.live_server_url)
        self.add_list_item("Reticulate splines")  ❶
        self.add_list_item("Immanentize eschaton")
        first_list_url = self.browser.current_url

        # She notices a "My lists" link, for the first time.
        self.browser.find_element(By.LINK_TEXT, "My lists").click()

        # She sees her email is there in the page heading
        self.wait_for(
            lambda: self.assertIn(
                "edith@example.com",
                self.browser.find_element(By.CSS_SELECTOR, "h1").text,
            )
        )

        # And she sees that her list is in there,
        # named according to its first list item
        self.wait_for(
            lambda: self.browser.find_element(By.LINK_TEXT, "Reticulate splines")
        )
        self.browser.find_element(By.LINK_TEXT, "Reticulate splines").click()
        self.wait_for(
            lambda: self.assertEqual(self.browser.current_url, first_list_url)
        )
```

❶ We'll define this `add_list_item()` shortly.

As you can see, we create a list with a couple of items. Then, we check that this list appears on a new "My lists" page, and that it's "named" after the first item in the list.

Let's validate that it really works by creating a second list, and seeing that appear on the "My lists" page as well. The FT continues, and while we're at it, we check that only logged-in users can see the "My lists" page:

src/functional_tests/test_my_lists.py (ch24l002)

```
[...]
self.wait_for(
    lambda: self.assertEqual(self.browser.current_url, first_list_url)
)

# She decides to start another list, just to see
self.browser.get(self.live_server_url)
self.add_list_item("Click cows")
second_list_url = self.browser.current_url

# Under "my lists", her new list appears
self.browser.find_element(By.LINK_TEXT, "My lists").click()
self.wait_for(lambda: self.browser.find_element(By.LINK_TEXT, "Click cows"))
self.browser.find_element(By.LINK_TEXT, "Click cows").click()
self.wait_for(
    lambda: self.assertEqual(self.browser.current_url, second_list_url)
)

# She logs out.  The "My lists" option disappears
self.browser.find_element(By.CSS_SELECTOR, "#id_logout").click()
self.wait_for(
    lambda: self.assertEqual(
        self.browser.find_elements(By.LINK_TEXT, "My lists"),
        [],
    )
)
```

Our FT uses a new helper method, `add_list_item()`, which abstracts away the process of entering text into the right input box. We define it in *base.py*:

src/functional_tests/base.py (ch24l003)

```
from selenium.webdriver.common.keys import Keys
[...]

    def add_list_item(self, item_text):
        num_rows = len(self.browser.find_elements(By.CSS_SELECTOR, "#id_list_table tr"))
        self.get_item_input_box().send_keys(item_text)
        self.get_item_input_box().send_keys(Keys.ENTER)
        item_number = num_rows + 1
        self.wait_for_row_in_list_table(f"{item_number}: {item_text}")
```

And while we're at it, we can use it in a few of the other FTs—like this, for example:

src/functional_tests/test_layout_and_styling.py (ch24l004-2)

```
        # She starts a new list and sees the input is nicely
        # centered there too
-       inputbox.send_keys("testing")
-       inputbox.send_keys(Keys.ENTER)
-       self.wait_for_row_in_list_table("1: testing")
+       self.add_list_item("testing")
+
```

I think it makes the FTs a lot more readable. I made a total of six changes—see if you agree with me.

Let's do a quick run of all FTs, a commit, and then back to the FT we're working on. The first error should look like this:

```
$ python src/manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: My lists; [...]
```

The Outside Layer: Presentation and Templates

The test is currently failing because it can't find a link saying "My lists". We can address that at the presentation layer, in *base.html*, in our navigation bar. Here's the minimal code change:

src/lists/templates/base.html (ch24l005)

```
    <nav class="navbar">
      <div class="container-fluid">
        <a class="navbar-brand" href="/">Superlists</a>
        {% if user.email %}
          <a class="navbar-link" href="#">My lists</a>
          <span class="navbar-text">Logged in as {{ user.email }}</span>
          <form method="POST" action="{% url 'logout' %}">
            [...]
```

Of course the `href="#"` means that link doesn't actually go anywhere, but it *does* get our FT along to the next failure:

```
$ python src/manage.py test functional_tests.test_my_lists
[...]
    lambda: self.assertIn(
            ~~~~~~~~~~~~~^
        "edith@example.com",
        ^^^^^^^^^^^^^^^^^^^
        self.browser.find_element(By.CSS_SELECTOR, "h1").text,
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    )
    ^
AssertionError: 'edith@example.com' not found in 'Your To-Do list'
```

That is telling us that we're going to have to build a page that at least has the user's email in its header. Let's start with the basics—a URL and a placeholder template for it. Again, we can go outside-in, starting at the presentation layer with just the URL and nothing else:

src/lists/templates/base.html (ch24l006)

```
{% if user.email %}
  <a class="navbar-link" href="{% url 'my_lists' user.email %}">My lists</a>
```

Moving Down One Layer to View Functions (the Controller)

That will cause a template error in the FT:

```
$ ./src/manage.py test functional_tests.test_my_lists
[...]
Internal Server Error: /
[...]
  File "...goat-book/src/lists/views.py", line 8, in home_page
    return render(request, "home.html", {"form": ItemForm()})
[...]
django.urls.exceptions.NoReverseMatch: Reverse for 'my_lists' not found.
'my_lists' is not a valid view function or pattern name.
[...]
ERROR: test_logged_in_users_lists_are_saved_as_my_lists [...]
[...]
selenium.common.exceptions.NoSuchElementException: [...]
```

To fix it, we'll need to start moving from working at the presentation layer, gradually into the controller layer—Django's URLs and views. As always, we start with a test. In this layer, a unit test is the way to go:

src/lists/tests/test_views.py (ch24l007)

```
class MyListsTest(TestCase):
    def test_my_lists_url_renders_my_lists_template(self):
        response = self.client.get("/lists/users/a@b.com/")
        self.assertTemplateUsed(response, "my_lists.html")
```

That gives:

```
$ python src/manage.py test lists
[...]
AssertionError: No templates used to render the response
```

That's because the URL doesn't exist yet, and a 404 has no template. Let's start our fix in *urls.py*:

src/lists/urls.py (ch24l008)

```
urlpatterns = [
    path("new", views.new_list, name="new_list"),
    path("<int:list_id>/", views.view_list, name="view_list"),
    path("users/<str:email>/", views.my_lists, name="my_lists"),
]
```

That gives us a new test failure, which informs us of what we should do. As you can see, it's pointing us at a *views.py*. We're clearly in the controller layer:

```
    path("users/<str:email>/", views.my_lists, name="my_lists"),
                               ^^^^^^^^^^^^^^
AttributeError: module 'lists.views' has no attribute 'my_lists'
```

Let's create a minimal placeholder then:

src/lists/views.py (ch24l009)

```
def my_lists(request, email):
    return render(request, "my_lists.html")
```

Let's also create a minimal template, with no real content except for the header that shows the user's email address:

src/lists/templates/my_lists.html (ch24l010)

```
{% extends 'base.html' %}

{% block header_text %}{{ user.email }}'s Lists{% endblock %}
```

That gets our unit tests passing:

```
$ ./src/manage.py test lists
[...]
OK
```

And hopefully it will address the current error in our FT:

```
$ python src/manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Reticulate splines; [...]
```

Step by step! Sure enough, the FT gets a little further. It can now find the email in the `<h1>`, but it's now saying that the "My lists" page doesn't yet show any lists. It wants them to appear as clickable links, named after the first item.

Another Pass, Outside-In

At each stage, we're still letting the FT drive what development we do. Starting again at the outside layer, in the template, we begin to write the template code we'd like to use to get the "My lists" page to work the way we want it to. As we do so, we start to specify the API we want from the code at the layers below.

A Quick Restructure of Our Template Composition

Let's take a look at our base template, *base.html*. It currently has a lot of content that's specific to editing to-do lists, which our "My lists" page doesn't need:

src/lists/templates/base.html

```
<div class="container">

  <nav class="navbar">
    [...]
  </nav>

  {% if messages %}
    [...]
  {% endif %}

  <div class="row justify-content-center p-5 bg-body-tertiary rounded-3">
    <div class="col-lg-6 text-center">
      <h1 class="display-1 mb-4">{% block header_text %}{% endblock %}</h1>

      <form method="POST" action="{% block form_action %}{% endblock %}" >  ❶
        [...]
      </form>
    </div>
  </div>

  <div class="row justify-content-center">
    <div class="col-lg-6">
      {% block table %}  ❷
      {% endblock %}
    </div>
  </div>

</div>

<script src="/static/lists.js"></script>  ❸
  [...]
```

❶ The `<form>` tag is definitely something we only want on pages where we edit lists. Everything else up to this point is generic enough to be on any page.

❷ Similarly, the `{% block table %}` isn't something we'd need on the "My lists" page.

❸ Finally, the `<script>` tag is specific to lists too.

So, we'll want to change things so that *base.html* is a bit more generic.

Let's recap. We've got three actual pages we want to render:

1. The home page (where you can enter a first to-do item to create a new list)
2. The "List" page (where you can view an existing list and add to it)
3. The "My lists" page (which is a list of all your existing lists)

And the home page and list page both share the same "form" elements and the *lists.js* JavaScript. But the "List" page is the only one that needs to show the full table of list items. The "My lists" page doesn't need anything related to editing or displaying lists.

So, we have some things shared between all three, and some only shared between the first and second.

So far, we've been using inheritance to share the common parts of our templates, but this is a good place to start using composition instead. At the moment, we're saying that "home" is a type of "base" template, but with the "table" section switched off, which is a bit awkward. Let's not make it even more awkward by saying that "list" is a "base" template with both the form and the table switched off! It might make more sense to say that "home" is a type of base template that includes a list form, but no table, and that "list" includes both the list form and the list table.

People often say "prefer composition over inheritance", because inheritance can become hard to reason about as the inheritance hierarchy grows. Composition is more flexible and often makes more sense. For a lengthy discussion of this topic, see Hynek Schlawack's definitive article on subclassing in Python.

So, let's do the following:

1. Pull out the `<form>` tag and the *lists.js* `<script>` tag into into some blocks we can "include" in our home page and lists page.
2. Move the `<table>` block so it only exists in the list page.
3. Take all the list-specific stuff out of the *base.html* template, making it into a more generic page with a header and a placeholder for generic content.

We'll use what's called an `include` to compose reusable template fragments when we don't want to use inheritance.

An Early Return So We're Refactoring Against Green

Before we start refactoring, let's put an early return in our FT, so we're refactoring against green tests:

src/functional_tests/test_my_lists.py (ch24l010-0)

```
        # She sees her email is there in the page heading
        self.wait_for(
            lambda: self.assertIn(
                "edith@example.com",
                self.browser.find_element(By.CSS_SELECTOR, "h1").text,
            )
        )
        return # TODO: resume here after templates refactor

        # And she sees that her list is in there,
        # named according to its first list item
        [...]
```

Verify the FTs are all green:

```
Ran 8 tests in 19.712s

OK
```

Factoring Out Two Template includes

First let's pull out the form and the script tag from *base.html*:

src/lists/templates/base.html (ch24l010-1)

```
@@ -58,43 +58,19 @@
         <div class="col-lg-6 text-center">
           <h1 class="display-1 mb-4">{% block header_text %}{% endblock %}</h1>

-          <form method="POST" action="{% block form_action %}{% endblock %}" >
-            {% csrf_token %}
-            <input
-              id="id_text"
-              name="text"
-              class="form-control
-                     form-control-lg
-                     {% if form.errors %}is-invalid{% endif %}"
-              placeholder="Enter a to-do item"
-              value="{{ form.text.value }}"
-              aria-describedby="id_text_feedback"
-              required
-            />
-            {% if form.errors %}
-              <div id="id_text_feedback" class="invalid-feedback">
-                {{ form.errors.text.0 }}
-              </div>
-            {% endif %}
-          </form>
+          {% block extra_header %}
+          {% endblock %}
+
         </div>
       </div>

-      <div class="row justify-content-center">
-        <div class="col-lg-6">
-          {% block table %}
-          {% endblock %}
-        </div>
-      </div>
+      {% block content %}
+      {% endblock %}

     </div>

-    <script src="/static/lists.js"></script>
-    <script>
-      window.onload = () => {
-        initialize("#id_text");
-      };
-    </script>
+    {% block scripts %}
+    {% endblock %}

   </body>
 </html>
```

You can see we've replaced all the list-specific stuff with three new blocks:

1. `extra_header` for anything we want to put in the big header section
2. `content` for the main content of the page
3. `scripts` for any JavaScript we want to include

Let's paste the `<form>` tag into a file at *src/lists/templates/includes/form.html* (having a subfolder in templates for includes is a common practice):

src/lists/templates/includes/form.html (ch24l010-2)

```
<form method="POST" action="{{ form_action }}" >  ❶
  {% csrf_token %}
  <input
    id="id_text"
    name="text"
    class="form-control
           form-control-lg
           {% if form.errors %}is-invalid{% endif %}"
    placeholder="Enter a to-do item"
    value="{{ form.text.value | default:'' }}"
    aria-describedby="id_text_feedback"
    required
  />
  {% if form.errors %}
    <div id="id_text_feedback" class="invalid-feedback">
      {{ form.errors.text.0 }}
    </div>
  {% endif %}
</form>
```

❶ This is the only change; we've replaced the `{% block form_action %}` with `{{ form_action }}`.

Let's paste the `script` tags verbatim into a new file at *includes/scripts.html*:

src/lists/templates/includes/scripts.html (ch24l010-3)

```
<script src="/static/lists.js"></script>

<script>
  window.onload = () => {
    initialize("#id_text");
  };
</script>
```

Now let's look at how to use the `include`, and how the `form_action` change plays out in the changes to *home.html*:

src/lists/templates/home.html (ch24l010-4)

```
{% extends 'base.html' %}

{% block header_text %}Start a new To-Do list{% endblock %}

{% block extra_header %}
  {% url 'new_list' as form_action %}  ❶
  {% include "includes/form.html" with form=form form_action=form_action %}  ❷
{% endblock %}

{% block scripts %}  ❸
  {% include "includes/scripts.html" %}
{% endblock %}
```

❶ The `{% url ... as %}` syntax lets us define a template variable inline.

❷ Then we use `{% include ... with key=value... %}` to pull in the contents of the `form.html` template, with the appropriate context variables passed in—a bit like calling a function.[1]

❸ The `scripts` block is just a straightforward `include` with no variables.

1 Strictly speaking, you could have omitted the `with=` in this case, as included templates automatically get the context of their parent. But sometimes you want to pass a context variable under a different name, so I like the `with`, for consistency and explicitness.

Now let's see it in *list.html*:

src/lists/templates/list.html (ch24l010-5)

```
@@ -2,12 +2,24 @@

 {% block header_text %}Your To-Do list{% endblock %}

-{% block form_action %}{% url 'view_list' list.id %}{% endblock %}

-{% block table %}
+{% block extra_header %}  ❶
+  {% url 'view_list' list.id as form_action %}
+  {% include "includes/form.html" with form=form form_action=form_action %}
+{% endblock %}
+
+{% block content %}  ❷
+<div class="row justify-content-center">
+  <div class="col-lg-6">
     <table class="table" id="id_list_table">
       {% for item in list.item_set.all %}
         <tr><td>{{ forloop.counter }}: {{ item.text }}</td></tr>
       {% endfor %}
     </table>
+  </div>
+</div>
+{% endblock %}
+
+{% block scripts %}  ❸
+  {% include "includes/scripts.html" %}
 {% endblock %}
```

❶ The `block table` becomes an `extra_header` block, and we use the `include` to pull in the form.

❷ The `block table` becomes a `content` block, with all the HTML we need for our table.

❸ And the `scripts` block is the same as the one from *home.html*.

Now a little rerun of all our FTs to make sure we haven't broken anything:

```
Ran 8 tests in 19.712s

OK
```

OK, let's remove the early return:

src/functional_tests/test_my_lists.py (ch24l010-6)

```
@@ -44,7 +44,6 @@ class MyListsTest(FunctionalTest):
                 self.browser.find_element(By.CSS_SELECTOR, "h1").text,
             )
         )
-        return # TODO: resume here after templates refactor

         # And she sees that her list is in there,
         # named according to its first list item
```

And we'll commit that as a nice refactor:

```
$ git add src/lists/templates
$ git commit -m "refactor templates to use composition/includes"
```

Now let's get back to our outside-in process, and to working in our template to drive out the requirements for our views layer.

Designing Our API Using the Template

With the early return removed, our FT is back to telling us that we need to actually show our lists—named after their first items—on the new "My lists" page:

```
$ ./src/manage.py test functional_tests.test_my_lists
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Reticulate splines; [...]
```

(If you haven't taken a look around the site recently, it does look pretty blank—see Figure 24-3.)

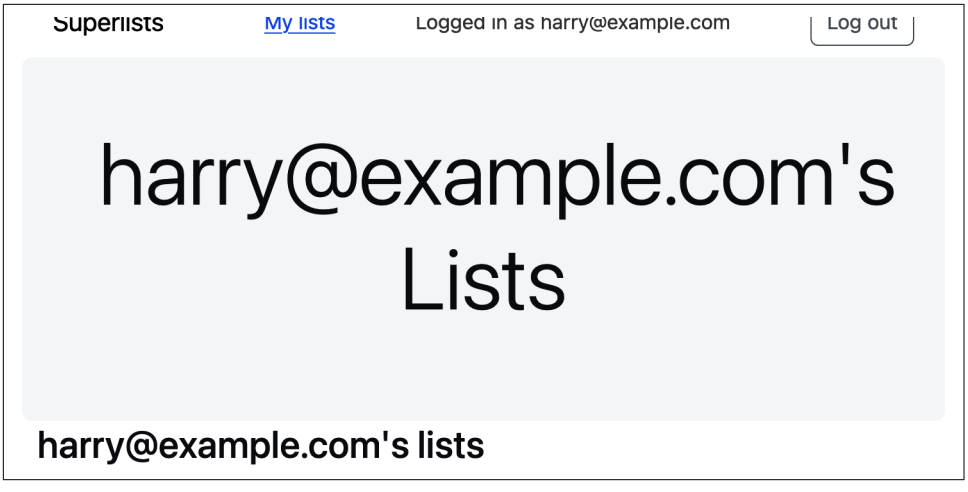


Figure 24-3. Not much to see here

So, in *my_lists.html*, we can now work in the `content` block:

src/lists/templates/my_lists.html (ch24l010-7)

```
[...]

{% block content %}
  <h2>{{ owner.email }}'s lists</h2>  ❶
  <ul>
    {% for list in owner.lists.all %}  ❷
      <li><a href="{{ list.get_absolute_url }}">{{ list.name }}</a></li>  ❸
    {% endfor %}
  </ul>
{% endblock %}
```

We've made several design decisions in this template that are going to filter their way down through the code:

❶ We want a variable called `owner` to represent the user in our template. This is what will allow one user to view another user's lists.

❷ We want to be able to iterate through the lists created by that user using `owner.lists.all`. (I happen to know how to make this work with the Django ORM.)

❸ We want to use `list.name` to print out the "name" of the list, which is currently specified as the text of its first element.

Programming by Wishful Thinking Again, Still

The phrase "programming by wishful thinking" was first popularised by the amazing, mind-expanding textbook Structure and Interpretation of Computer Programs (SICP), which I *cannot* recommend highly enough.

In it, the authors use it as a way to think about and write code at a higher level of abstraction, without worrying about the details of a lower level that might not even exist yet. For them, it's a key tool for designing programs and managing complexity.

We've been doing a lot of "programming by wishful thinking" in this book. We've talked about how TDD itself is a form of wishful thinking; our tests express that we *wish* we had code that worked in such-and-such a way.

Outside-in TDD is very much an extension of this philosophy. We start writing code at the higher levels based on what we *wish* we had at the lower levels, even though it doesn't exist yet…

YAGNI also comes into it. By driving our development from the outside in, each piece of code we write is only there because we know it's actually needed by a higher layer and, ultimately, by the user.

We can rerun our FTs to check that we didn't break anything, and to see whether we've gotten any further:

```
$ python src/manage.py test functional_tests
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: Reticulate splines; [...]

 ----------------------------------------------------------------------
Ran 8 tests in 77.613s

FAILED (errors=1)
```

Well, no further—but at least we didn't break anything. Time for a commit:

```
$ git add src/lists
$ git diff --staged   # urls+views.py, templates
$ git commit -m "url, placeholder view, and first-cut templates for my_lists"
```

Moving Down to the Next Layer: What the View Passes to the Template

Now our views layer needs to respond to the requirements we've laid out in the template layer, by giving it the objects it needs—in this case, the list owner:

src/lists/tests/test_views.py (ch24l011)

```
from accounts.models import User
[...]


class MyListsTest(TestCase):
    def test_my_lists_url_renders_my_lists_template(self):
        [...]

    def test_passes_correct_owner_to_template(self):
        User.objects.create(email="wrong@owner.com")
        correct_user = User.objects.create(email="a@b.com")
        response = self.client.get("/lists/users/a@b.com/")
        self.assertEqual(response.context["owner"], correct_user)
```

That gives:

```
KeyError: 'owner'
```

So:

src/lists/views.py (ch24l012)

```
from accounts.models import User
[...]


def my_lists(request, email):
    owner = User.objects.get(email=email)
    return render(request, "my_lists.html", {"owner": owner})
```

That gets our new test passing, but we'll also see an error from the previous test. We just need to add a user for it as well:

src/lists/tests/test_views.py (ch24l013)

```
    def test_my_lists_url_renders_my_lists_template(self):
        User.objects.create(email="a@b.com")
        [...]
```

And we get to an OK:

```
OK
```

The Next "Requirement" from the Views Layer: New Lists Should Record Owner

Before we move down to the model layer, there's another part of the code at the view layer that will need to use our model: we need some way for newly created lists to be assigned to an owner, if the current user is logged in to the site.

Here's a first crack at writing the test:

src/lists/tests/test_views.py (ch24l014)

```
class NewListTest(TestCase):
    [...]

    def test_list_owner_is_saved_if_user_is_authenticated(self):
        user = User.objects.create(email="a@b.com")
        self.client.force_login(user)  ❶
        self.client.post("/lists/new", data={"text": "new item"})
        new_list = List.objects.get()
        self.assertEqual(new_list.owner, user)
```

❶ `force_login()` is the way you get the test client to make requests with a logged-in user.

The test fails as follows:

```
AttributeError: 'List' object has no attribute 'owner'
```

To fix it, let's first try writing code like this:

src/lists/views.py (ch24l015)

```
def new_list(request):
    form = ItemForm(data=request.POST)
    if form.is_valid():
        nulist = List.objects.create()
        nulist.owner = request.user  ❶
        nulist.save()  ❷
        form.save(for_list=nulist)
        return redirect(nulist)
    else:
        return render(request, "home.html", {"form": form})
```

❶ We'll set the `.owner` attribute on our new list.

❷ And we'll try and save it to the database.

But it won't actually work, because we don't know *how* to save a list owner yet:

```
        self.assertEqual(new_list.owner, user)
                         ^^^^^^^^^^^^^^
    AttributeError: 'List' object has no attribute 'owner'
```

A Decision Point: Whether to Proceed to the Next Layer with a Failing Test

In order to get this test passing, as it's written now, we have to move down to the model layer. However, it means doing more work with a failing test, which is not ideal. The alternative is to rewrite the test to make it more *isolated* from the level below, using mocks.

On the one hand, it's a lot more effort to use mocks, and it can lead to tests that are harder to read. On the other hand, advocates of London-school TDD are very keen on the approach. You can read an exploration of this approach in Online Appendix: Test Isolation and "Listening to Your Tests".

For now, we'll accept the trade-off: moving down one layer with failing tests, but avoiding the extra mocks.

Let's do a commit, and then `tag` the commit as a way of remembering our position if we want to revisit this decision later:

```
$ git commit -am "new_list view tries to assign owner but cant"
$ git tag revisit_this_point_with_isolated_tests
```

Moving Down to the Model Layer

Our outside-in design has driven out two requirements for the model layer: we want to be able to assign an owner to a list using the attribute `.owner`, and we want to be able to access the list's owner with the API `owner.lists.all()`.

Let's write a test for that:

src/lists/tests/test_models.py (ch24l018)

```
from accounts.models import User
[...]


class ListModelTest(TestCase):
    def test_get_absolute_url(self):
        [...]
    def test_list_items_order(self):
        [...]

    def test_lists_can_have_owners(self):
        user = User.objects.create(email="a@b.com")
        mylist = List.objects.create(owner=user)
        self.assertIn(mylist, user.lists.all())
```

And that gives us a new unit test failure:

```
        mylist = List.objects.create(owner=user)
        [...]
TypeError: List() got unexpected keyword arguments: 'owner'
```

The naive implementation would be this:

```
from django.conf import settings
[...]

class List(models.Model):
    owner = models.ForeignKey(settings.AUTH_USER_MODEL)
```

But we want to make sure the list owner is optional. Explicit is better than implicit, and tests are documentation, so let's have a test for that too:

src/lists/tests/test_models.py (ch24l020)

```
    def test_list_owner_is_optional(self):
        List.objects.create()  # should not raise
```

The correct implementation is this:

src/lists/models.py (ch24l021)

```
class List(models.Model):
    owner = models.ForeignKey(
        "accounts.User",
        related_name="lists",
        blank=True,
        null=True,
        on_delete=models.CASCADE,
    )

    def get_absolute_url(self):
        return reverse("view_list", args=[self.id])
```

Now running the tests gives the usual database error:

```
    return super().execute(query, params)
           ~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^
django.db.utils.OperationalError: table lists_list has no column named owner_id
```

Because we need to make some migrations:

```
$ python src/manage.py makemigrations
Migrations for 'lists':
  src/lists/migrations/0007_list_owner.py
    + Add field owner to list
```

We're almost there; a couple more failures in some of our old tests:

```
ERROR: test_can_save_a_POST_request
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x1069852e>>": "List.owner"
must be a "User" instance.
[...]

ERROR: test_redirects_after_POST
[...]
ValueError: Cannot assign "<SimpleLazyObject:
<django.contrib.auth.models.AnonymousUser object at 0x106a1b440>>": "List.owner"
must be a "User" instance.
```

We're moving back up to the views layer now, just doing a little tidying up. Notice that these are in the existing test for the `new_list` view, when we haven't got a logged-in user.

The tests are reminding us to think of this use case too: we should only save the list owner when the user is actually logged in. The `.is_authenticated` attribute we came across in Chapter 19 comes in useful now:[2]

src/lists/views.py (ch24l023)

```
    if form.is_valid():
        nulist = List.objects.create()
        if request.user.is_authenticated:
            nulist.owner = request.user
            nulist.save()
        form.save(for_list=nulist)
        return redirect(nulist)
        [...]
```

2 When they're not logged in, Django represents users using a class called `AnonymousUser`, whose `.is_authenticated` is always `False`.

And that gets us passing!

```
$ python src/manage.py test lists
[...]

Ran 36 tests in 0.237s

OK
```

This is a good time for a commit:

```
$ git add src/lists
$ git commit -m "lists can have owners, which are saved on creation."
```

Final Step: Feeding Through the .name API from the Template

The last thing our outside-in design wanted came from the templates, which want to be able to access a list "name" based on the text of its first item:

src/lists/tests/test_models.py (ch24l024)

```
def test_list_name_is_first_item_text(self):
    list_ = List.objects.create()
    Item.objects.create(list=list_, text="first item")
    Item.objects.create(list=list_, text="second item")
    self.assertEqual(list_.name, "first item")
```

src/lists/models.py (ch24l025)

```
@property
def name(self):
    return self.item_set.first().text
```

And that, believe it or not, actually gets us a passing test and a working "My lists" page (see Figure 24-4)!

```
$ python src/manage.py test functional_tests
[...]
Ran 8 tests in 93.819s

OK
```

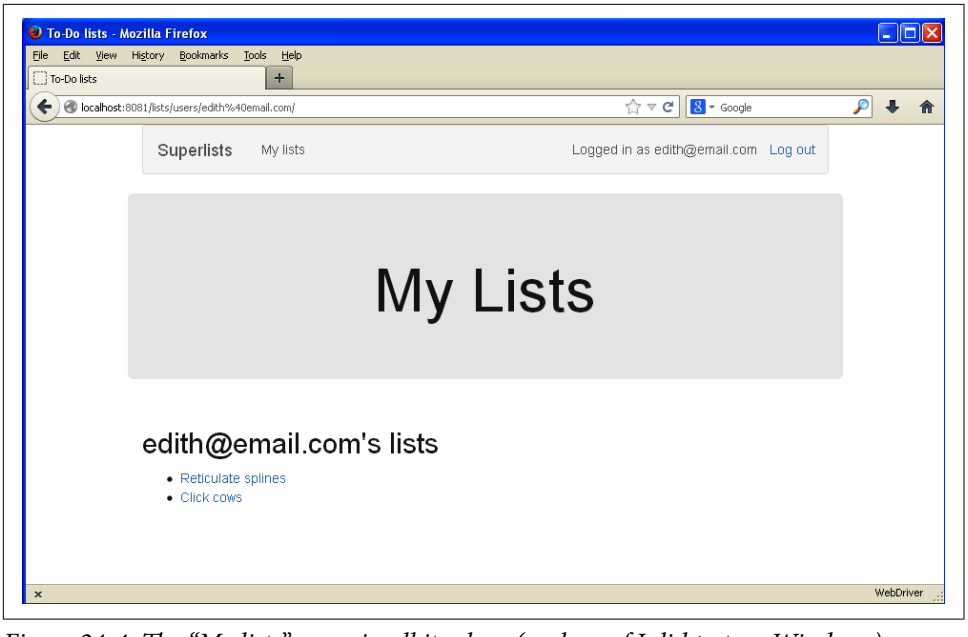


Figure 24-4. The "My lists" page, in all its glory (and proof I did test on Windows)

The @property Decorator in Python

If you haven't seen it before, the `@property` decorator transforms a method on a class to make it appear to the outside world like an attribute.

This is a powerful feature of the language, because it makes it easy to implement "duck typing"—to change the implementation of a property without changing the interface of the class. In other words, if we decide to change `.name` into being a "real" attribute on the model, stored as text in the database, then we will be able to do so entirely transparently—as far as the rest of our code is concerned, will still be able to just access `.name` and get the list name, without needing to know about the implementation.

Raymond Hettinger gave a classic, beginner-friendly talk on this topic at PyCon back in 2013, which I enthusiastically recommend (it covers about a million good practices for Pythonic class design besides). Of course, in the Django template language, `.name` would still call the method even if it didn't have `@property`, but that's a particularity of Django, and doesn't apply to Python in general…

In the next chapter, it's time to recruit some computers to do more of the work for us. Let's talk about continuous integration (CI).

Outside-In TDD

Outside-in TDD
: This is methodology for building code, driven by tests, which proceeds by starting from the "outside" layers (presentation, GUI), and moving "inwards" step by step, via view/controller layers, down towards the model layer. The idea is to drive the design of your code from how it will be used, rather than trying to anticipate requirements from the bottom up.

Programming by wishful thinking
: The outside-in process is sometimes called "programming by wishful thinking". Actually, any kind of TDD involves some wishful thinking. We're always writing tests for things that don't exist yet.

The pitfalls of outside-in
: Outside-in isn't a silver bullet. It encourages us to focus on things that are immediately visible to the user, but it won't automatically remind us to write other critical tests that are less user-visible—things like security, for example. You'll need to remember them yourself.

CHAPTER 25

CI: Continuous Integration

As our site grows, it takes longer and longer to run all of our functional tests. If this continues, the danger is that we're going to stop bothering.

Rather than let that happen, we can automate the running of functional tests by setting up "continuous integration", or CI. That way, in day-to-day development, we can just run the FT that we're working on at that time, and rely on CI to run all the other tests automatically and let us know if we've broken anything accidentally.

The unit tests should stay fast enough that we can keep running the full suite locally, every few seconds.

Continuous integration is another practice that was popularised by Kent Beck's extreme programming (XP) movement in the 1990s.

As we'll see, one of the great frustrations of configuring CI is that the feedback loop is so much slower than working locally. As we go along, we'll look for ways to optimise for that where we can.

While debugging, we'll also touch on the theme of *reproducibility*. It's the idea that we want to be able to reproduce behaviours of our CI environment locally—in the same way that we try and make our production and dev environments as similar as possible.

CI in Modern Development Workflows

We use CI for a number of reasons:

- As mentioned, it can patiently run the full suite of tests, even if they've grown too large to run locally.
- It can act as a "gate" in your deployment/release process, to ensure that you never deploy code that isn't passing all the tests.
- In open source projects that use a "pull request" workflow, it's a way to ensure that any code submitted by potentially unknown contributors passes all your tests, before you consider merging it.
- It's (sadly) increasingly common in corporate environments to see this pull request process and its associated CI checks to be used as the default way for teams to merge all code changes.[1]

Choosing a CI Service

In the early days, CI would be implemented by configuring a server (perhaps under a desk in the corner of the office) with software on it that could pull down all the code from the main branch at the end of each day, and scripts to compile all the code and run all the tests—a process that became known as a "build". Then, each morning, developers would take a look at the results, and deal with any broken builds.

As the practice spread, and feedback cycles grew faster, CI software matured. CI has become a common cloud-based service, designed to integrate with code hosting providers like GitHub—or even provided directly by the same providers. GitHub has "GitHub Actions", and because it's like, *right there*, it's probably the most popular choice for open source projects these days. In a corporate environment, you might come across other solutions like CircleCI, Travis CI, and GitLab.

It is still absolutely possible to download and self-host your own CI server; in the first and second editions of this book, I demonstrated the use of Jenkins, a popular tool at the time. But the installation and subsequent admin/maintenance burden is not effort-free, so for this edition I wanted to pick a service more like the kind of thing you're likely to encounter in your day job—while trying not to endorse the largest commercial provider. There's nothing wrong with GitHub Actions! It just doesn't need any *more* help dominating the market.

1 I say "sadly" because you *should* be able to trust your colleagues, not put them through a process designed for open source projects to de-risk code contributions from random strangers on the internet. Look up "trunk-based development" if you want to see more old people shouting at clouds on this topic.

So I've decided to use GitLab in this book. It is absolutely a commercial service, but it retains an open source version, and you can self-host it if you want to. The syntax (it's always YAML…) and core concepts are common across all providers, so the things you learn here will be replicable in whichever service you encounter in future.

Like most of the services out there, GitLab has a free tier, which will work fine for our purposes.

Getting Our Code into GitLab

GitLab is primarily a code hosting service, like GitHub, so the first thing to do is get our code up there.

Signing Up

Head over to *GitLab.com*, and sign up for a free account.

Then, head over to your profile page, and find the SSH Keys section, and upload a copy of your public key.

Starting a Project

Then, use the New Project → Create Blank Project option, as in Figure 25-1. Feel free to name the project whatever you want; you can see I've fancifully named mine with a "z". I'm a free spirit, what can I say.

Figure 25-1. Creating a new repo on GitLab

Pushing Our Code Up Using Git Push

First, we set up GitLab as a "remote" for our project:

```
# substitute your username and project name as necessary
$ git remote add gitlab git@gitlab.com:yourusername/superlists.git
$ git remote -v
gitlab    git@gitlab.com:hjwp/superlistz.git (fetch)
gitlab    git@gitlab.com:hjwp/superlistz.git (push)
origin    git@github.com:hjwp/book-example.git (fetch)
origin    git@github.com:hjwp/book-example.git (push)
# (as you can see i already had a remote for github called origin)
```

Now we can push up our code with `git push`:

```
$ git push gitlab
Enumerating objects: 706, done.
Counting objects: 100% (706/706), done.
Delta compression using up to 11 threads
Compressing objects: 100% (279/279), done.
Writing objects: 100% (706/706), 918.72 KiB | 131.25 MiB/s, done.
Total 706 (delta 413), reused 682 (delta 408), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (413/413), done.
To gitlab.com:hjwp/superlistz.git
 * [new branch]        main -> main
branch main set up to track gitlab/main.
```

If you refresh the GitLab UI, you should now see your code, as in Figure 25-2.

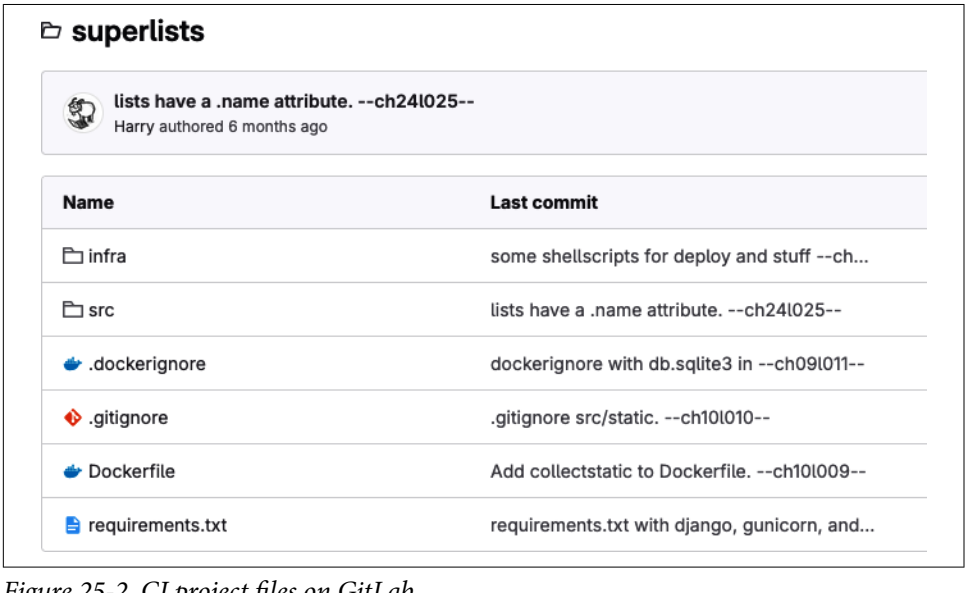


Figure 25-2. CI project files on GitLab

Setting Up a First Cut of a CI Pipeline

The "pipeline" terminology was popularised by Dave Farley and Jez Humble in their book *Continuous Delivery* (Addison-Wesley). The name alludes to the fact that a CI build typically has a series, where the process flows from one to another.

Go to Build → Pipelines, and you'll see a list of example templates. When getting to know a new configuration language, it's nice to be able to start with something that works, rather than a blank slate. I chose the Python example template and made a few customisations, but you could just as easily start from a blank slate and paste what I have here (YAML, once again, folks!):

.gitlab-ci.yml (ch25l001)

```
# Use the same image as our Dockerfile
image: python:slim

# These two settings let us cache pip-installed packages,
# it came from the default template
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
cache:
  paths:
    - .cache/pip

# "setUp" phase, before the main build
before_script:
  - python --version ; pip --version  # For debugging
  - pip install virtualenv
  - virtualenv .venv
  - source .venv/bin/activate

# This is the main build
test:
  script:
    - pip install -r requirements.txt  ❶
    # unit tests
    - python src/manage.py test lists accounts  ❷
    # (if those pass) all tests, incl. functional.
    - pip install selenium  ❸
    - cd src && python manage.py test  ❹
```

❶ We start by installing our core requirements.

❷ I've decided to run the unit tests first. This gives us an "early failure" if there's any problem at this stage, and saves us from having to run—and more importantly, wait for—the FTs to run.

❸ Then we need Selenium for the functional tests. Again, I'm delaying this `pip install` until it's absolutely necessary, to get feedback as quickly as possible.

❹ And here is a full test run, including the functional tests.

It's a good idea in CI pipelines to try and run the quickest tests first, so that you can get feedback as quickly as possible.

You can use the GitLab web UI to edit your pipeline YAML, and then when you save it, you can go check for results straight away.

But it is also just a file in your repo! So as we go on through the chapter, you can also just edit it locally. You'll need to commit it and then `git push` up to GitLab, and then go check the Jobs section in the Build UI to see the results of your changes:

```
$ git push gitlab
```

First Build! (and First Failure)

Whichever way you click through the UI, you should be able to find your way to see the output of the build job, as in Figure 25-3.

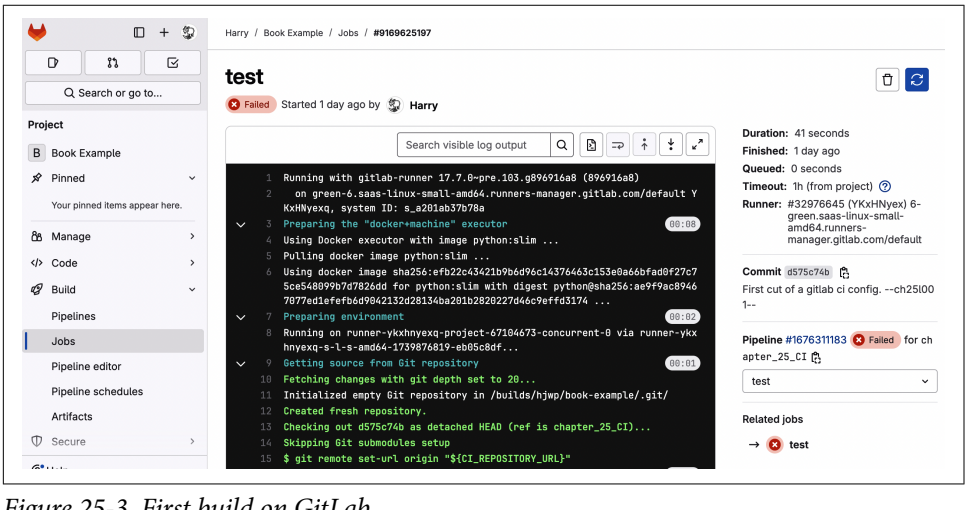


Figure 25-3. First build on GitLab

Here's a selection of what I saw in the output console:

```
Running with gitlab-runner 17.7.0~pre.103.g896916a8 (896916a8)
  on green-1.saas-linux-small-amd64.runners-manager.gitlab.com/default
  JLgUopmM, system ID: s_deaa2ca09de7
Preparing the "docker+machine" executor 00:20
Using Docker executor with image python:latest ...
Pulling docker image python:latest ...
[...]
$ python src/manage.py test lists accounts
Creating test database for alias 'default'...
Found 55 test(s).
System check identified no issues (0 silenced).
.................../builds/hjwp/book-example/.venv/lib/python3.14/site-packages/django/core
/handlers/base.py:61: UserWarning: No directory at: /builds/hjwp/book-example/src/static/
  mw_instance = middleware(adapted_handler)
.....................................
 ----------------------------------------------------------------------
Ran 53 tests in 0.129s
OK
Destroying test database for alias 'default'...
$ pip install selenium
Collecting selenium
  Using cached selenium-4.28.1-py3-none-any.whl.metadata (7.1 kB)
Collecting urllib3<3,>=1.26 (from urllib3[socks]<3,>=1.26->selenium)
[...]
Successfully installed attrs-25.1.0 certifi-2025.1.31 h11-0.14.0 idna-3.10
outcome-1.3.0.post0 pysocks-1.7.1 selenium-4.28.1 sniffio-1.3.1 sortedcontainers-2.4.0
trio-0.29.0 trio-websocket-0.12.1 typing_extensions-4.12.2 urllib3-2.3.0
websocket-client-1.8.0 wsproto-1.2.0
$ cd src && python manage.py test
Creating test database for alias 'default'...
Found 63 test(s).
System check identified no issues (0 silenced).
......../builds/hjwp/book-example/.venv/lib/python3.14/site-packages/django/core/handlers
/base.py:61: UserWarning: No directory at: /builds/hjwp/book-example/src/static/
  mw_instance = middleware(adapted_handler)
.............................................EEEEEEEE
======================================================================
ERROR: test_layout_and_styling (functional_tests.test_layout_and_styling.
LayoutAndStylingTest.test_layout_and_styling)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/builds/hjwp/book-example/src/functional_tests/base.py", line 30, in setUp
    self.browser = webdriver.Firefox()
                   ~~~~~~~~~~~~~~~~~^^

[...]
selenium.common.exceptions.WebDriverException: Message: Process unexpectedly closed with
status 255
 ----------------------------------------------------------------------
Ran 61 tests in 8.658s
FAILED (errors=8)

selenium.common.exceptions.WebDriverException: Message: Process unexpectedly closed with
status 255
```

If GitLab won't run your build at this point, you may need to go through some sort of identity-verification process. Check your profile page.

You can see we got through the unit tests, and then in the full test run we have 8 errors out of 63 tests. The FTs are all failing. I'm "lucky" because I've done this sort of thing many times before, so I know what to expect: it's failing because Firefox isn't installed in the image we're using.

Let's modify the script, and add an `apt install`. Again we'll do it as late as possible:

.gitlab-ci.yml (ch25l002)

```
# This is the main build
test:
  script:
    - pip install -r requirements.txt
    # unit tests
    - python src/manage.py test lists accounts
    # (if those pass) all tests, incl. functional.
    - apt update -y && apt install -y firefox-esr  ❶
    - pip install selenium
    - cd src && python manage.py test
```

❶ We use the Debian Linux `apt` package manager to install Firefox. `firefox-esr` is the "extended support release", which is a more stable version of Firefox to test against.

When you save that change (and commit and push if necessary), the pipeline will run again. If you wait a bit, you'll see we get a slightly different failure:

```
$ apt-get update -y && apt-get install -y firefox-esr
Get:1 http://deb.debian.org/debian bookworm InRelease [151 kB]
Get:2 http://deb.debian.org/debian bookworm-updates InRelease [55.4 kB]
Get:3 http://deb.debian.org/debian-security bookworm-security InRelease [48.0 kB]
[...]
The following NEW packages will be installed:
  adwaita-icon-theme alsa-topology-conf alsa-ucm-conf at-spi2-common
  at-spi2-core dbus dbus-bin dbus-daemon dbus-session-bus-common
  dbus-system-bus-common dbus-user-session dconf-gsettings-backend
  dconf-service dmsetup firefox-esr fontconfig fontconfig-config
[...]
Get:117 http://deb.debian.org/debian-security bookworm-security/main amd64
firefox-esr amd64 128.7.0esr-1~deb12u1 [69.8 MB]
[...]
Selecting previously unselected package firefox-esr.
Preparing to unpack .../105-firefox-esr_128.7.0esr-1~deb12u1_amd64.deb ...
Adding 'diversion of /usr/bin/firefox to /usr/bin/firefox.real by firefox-esr'
Unpacking firefox-esr (128.7.0esr-1~deb12u1) ...
[...]
Setting up firefox-esr (128.7.0esr-1~deb12u1) ...
update-alternatives: using /usr/bin/firefox-esr to provide
/usr/bin/x-www-browser (x-www-browser) in auto mode
[...]
======================================================================
ERROR: test_multiple_users_can_start_lists_at_different_urls
(functional_tests.test_simple_list_creation.NewVisitorTest.
test_multiple_users_can_start_lists_at_different_urls)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "/builds/hjwp/book-example/src/functional_tests/base.py", line 30, in setUp
    self.browser = webdriver.Firefox()
                   ~~~~~~~~~~~~~~~~~^^
[...]
selenium.common.exceptions.WebDriverException: Message: Process unexpectedly
closed with status 1
 ----------------------------------------------------------------------
Ran 61 tests in 3.654s
FAILED (errors=8)
```

We can see Firefox installing OK, but we still get an error. This time, it's exit code 1.

Trying to Reproduce a CI Error Locally

The cycle of "change *.gitlab-ci.yml*, push, wait for a build, check results" is painfully slow. Let's see if we can reproduce this error locally.

To reproduce the CI environment locally, I put together a quick Dockerfile, by copy-pasting the steps in the `script` section and prefixing them with `RUN` commands:

infra/Dockerfile.ci (ch25l003)

```
FROM python:slim

RUN pip install virtualenv
RUN virtualenv .venv

# this won't work
# RUN source .venv/bin/activate
# use full path to venv instead.

COPY requirements.txt requirements.txt
RUN .venv/bin/pip install -r requirements.txt
RUN apt update -y && apt install -y firefox-esr
RUN .venv/bin/pip install selenium

COPY infra/debug-ci.py debug-ci.py
CMD .venv/bin/python debug-ci.py
```

And let's add a little debug script at *debug-ci.py*:

infra/debug-ci.py (ch25l004)

```
from selenium import webdriver

# just try to open a selenium session
webdriver.Firefox().quit()
```

We build and run it like this:

```
$ docker build -f infra/Dockerfile.ci -t debug-ci . && \
  docker run -it debug-ci
[...]
 => [internal] load build definition from infra/Dockerfile.ci         0.0s
 => => transferring dockerfile: [...]
 => [internal] load metadata for docker.io/library/python:slim [...]
 => [1/8] FROM docker.io/library/python:slim@sha256:[...]
 => CACHED [2/8] RUN pip install virtualenv                           0.0s
 => CACHED [3/8] RUN virtualenv .venv                                 0.0s
 => CACHED [4/8] COPY requirements.txt requirements.txt               0.0s
 => CACHED [5/8] RUN .venv/bin/pip install -r requirements.txt        0.0s
 => CACHED [6/8] RUN apt update -y && apt install -y firefox-esr      0.0s
 => CACHED [7/8] RUN .venv/bin/pip install selenium                   0.0s
 => [8/8] COPY infra/debug-ci.py debug-ci.py                          0.0s
 => exporting to image                                                0.0s
 => => exporting layers                                               0.0s
 => => writing image sha256:[...]
 => => naming to docker.io/library/debug-ci                           0.0s
Traceback (most recent call last):
  File
  "//.venv/lib/python3.14/site-packages/selenium/webdriver/common/driver_finder.py",
  line 67, in _binary_paths
    output = SeleniumManager().binary_paths(self._to_args())
[...]
selenium.common.exceptions.WebDriverException: Message: Unsupported
platform/architecture combination: linux/aarch64

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "//debug-ci.py", line 4, in <module>
    webdriver.Firefox().quit()
    ~~~~~~~~~~~~~~~~~~^^
[...]
selenium.common.exceptions.NoSuchDriverException: Message: Unable to obtain
driver for firefox; For documentation on this error, please visit:
https://www.selenium.dev/documentation/webdriver/troubleshooting/errors/driver_location
```

You might not see this—that "Unsupported platform/architecture combination" error is spurious; it's because I was on a Mac. Let's try again with:

```
$ docker build -f infra/Dockerfile.ci -t debug-ci --platform=linux/amd64 . && \
  docker run --platform=linux/amd64 -it debug-ci
[...]
Traceback (most recent call last):
  File "//debug-ci.py", line 4, in <module>
    webdriver.Firefox().quit()
[...]
selenium.common.exceptions.WebDriverException: Message: Process unexpectedly
closed with status 1
```

OK, that's a reproduction of our issue. But no further clues yet!

Enabling Debug Logs for Selenium/Firefox/Webdriver

Getting debug information out of Selenium can be a bit fiddly. I tried two avenues: setting `options` and setting the `service`. The former doesn't really work as far as I can tell, but the latter does:

infra/debug-ci.py (ch25l005)

```
import subprocess

from selenium import webdriver

options = webdriver.FirefoxOptions()  ❶
options.log.level = "trace"

service = webdriver.FirefoxService(  ❷
    log_output=subprocess.STDOUT, service_args=["--log", "trace"]
)

# just try to open a selenium session
webdriver.Firefox(options=options, service=service).quit()
```

❶ This is how I attempted to increase the log level using `options`. I had to reverse-engineer it from the source code, and it doesn't seem to work anyway, but I thought I'd leave it here for future reference. There is some limited info in the Selenium docs.

❷ This is the `FirefoxService` config class, which *does* seem to let you print some debug info. I'm configuring it to print to standard output.

Sure enough, we can see some output now!

```
$ docker build -f infra/Dockerfile.ci -t debug-ci --platform=linux/amd64 . && \
  docker run --platform=linux/amd64 -it debug-ci
[...]
1234567890111   geckodriver     INFO    Listening on 127.0.0.1:XXXX
1234567890112   webdriver::server       DEBUG   -> POST /session
{"capabilities": {"firstMatch": [{}], "alwaysMatch": {"browserName": "firefox",
"acceptInsecureCerts": true, ... , "moz:firefoxOptions": {"binary":
"/usr/bin/firefox", "prefs": {"remote.active-protocols": 1}, "log": {"level":
"trace"}}}}}
1234567890111   geckodriver::capabilities       DEBUG   Trying to read firefox
version from ini files
1234567890111   geckodriver::capabilities       DEBUG   Trying to read firefox
version from binary
1234567890111   geckodriver::capabilities       DEBUG   Found version
128.10.1esr
1740029792102   mozrunner::runner       INFO    Running command:
MOZ_CRASHREPORTER="1" MOZ_CRASHREPORTER_NO_REPORT="1"
MOZ_CRASHREPORTER_SHUTDOWN="1" [...]
"--remote-debugging-port" [...]
"-no-remote" "-profile" "/tmp/rust_mozprofile[...]
1234567890111   geckodriver::marionette DEBUG   Waiting 60s to connect to
browser on 127.0.0.1
1234567890111   geckodriver::browser    TRACE   Failed to open
/tmp/rust_mozprofile[...]
1234567890111   geckodriver::marionette TRACE   Retrying in 100ms
Error: no DISPLAY environment variable specified
1234567890111   geckodriver::browser    DEBUG   Browser process stopped: exit
status: 1
1234567890112   webdriver::server       DEBUG   <- 500 Internal Server Error
{"value":{"error":"unknown error","message":"Process unexpectedly closed with
status 1","stacktrace":""}}
Traceback (most recent call last):
  File "//debug-ci.py", line 13, in <module>
    webdriver.Firefox(options=options, service=service).quit()
[...]
selenium.common.exceptions.WebDriverException: Message: Process unexpectedly
closed with status 1
```

Well, it wasn't immediately obvious what's going on there, but I did eventually get a clue from the line that says `no DISPLAY environment variable specified`.

Out of curiosity, I thought I'd try running `firefox` directly:[2]

```
$ docker build -f infra/Dockerfile.ci -t debug-ci --platform=linux/amd64 . && \
  docker run --platform=linux/amd64 -it debug-ci firefox
[...]
Error: no DISPLAY environment variable specified
```

Sure enough, the same error.

2 If you remember from Chapter 9, `docker run` by default runs the command specified in `CMD`, but you can override that by specifying a different command to run at the end of the parameter list.

Enabling Headless Mode for Firefox

If you search around for this error, you'll eventually find enough pointers to the answer: Firefox is crashing because it can't find a display. Servers are "headless", meaning they don't have a screen. Thankfully, Firefox has a headless mode, which we can enable by setting an environment variable, `MOZ_HEADLESS`.

Let's confirm that locally. We'll use the `-e` flag for `docker run`:

```
$ docker build -f infra/Dockerfile.ci -t debug-ci --platform=linux/amd64 . && \
  docker run -e MOZ_HEADLESS=1 --platform=linux/amd64 -it debug-ci
1234567890111   geckodriver     INFO    Listening on 127.0.0.1:43137
[...]
*** You are running in headless mode.
[...]
1234567890112   webdriver::server       DEBUG   Teardown [...]
1740030525996   Marionette      DEBUG   Closed connection 0
1234567890111   geckodriver::browser    DEBUG   Browser process stopped: exit
status: 0
1234567890112   webdriver::server       DEBUG   <- 200 OK [...]
```

It takes quite a long time to run, and there's lots of debug out, but…it looks OK! That's no longer an error.

Let's set that environment variable in our CI script:

.gitlab-ci.yml (ch25l006)

```
variables:
  # Put pip-cache in home folder so we can use gitlab cache
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
  # Make Firefox run headless.
  MOZ_HEADLESS: "1"
```

Using a local Docker image to reproduce the CI environment is a hint that it might be worth investing time in running CI in a custom Docker image that you fully control; this is another way of improving *reproducibility*. We won't have time to go into detail in this book though.

And we'll see what happens when we do `git push gitlab` again.

A Common Bugbear: Flaky Tests

Did it work for you? For me, it *almost* did. All but one of the FTs passed, but I did see one unexpected error:

```
+ python manage.py test functional_tests
......F.
======================================================================
FAIL: test_can_start_a_todo_list
(functional_tests.test_simple_list_creation.NewVisitorTest)
 ----------------------------------------------------------------------
Traceback (most recent call last):
  File "...goat-book/functional_tests/test_simple_list_creation.py", line
38, in test_can_start_a_todo_list
    self.wait_for_row_in_list_table('2: Use peacock feathers to make a fly')
  File "...goat-book/functional_tests/base.py", line 51, in
wait_for_row_in_list_table
    raise e
  File "...goat-book/functional_tests/base.py", line 47, in
wait_for_row_in_list_table
    self.assertIn(row_text, [row.text for row in rows])
AssertionError: '2: Use peacock feathers to make a fly' not found in ['1: Buy
peacock feathers']
 ----------------------------------------------------------------------
```

Now, you might not see this error, but it's common for the switch to CI to flush out some "flaky" tests—things that will fail intermittently. In CI, a common cause is the "noisy neighbour" problem, where the CI server might be much slower than your own machine, thus flushing out some race conditions—or in this case, just randomly hanging for a few seconds, taking us past the default timeout.

Let's give ourselves some tools to help debug though.

Taking Screenshots

To be able to debug unexpected failures that happen on a remote server, it would be good to see a picture of the screen at the moment of the failure, and maybe also a dump of the page's HTML.

We can do that using some custom logic in our FT class `tearDown`. We'll need to do a bit of introspection of `unittest` internals (a private attribute called `._outcome`) but this will work:[3]

3 …or at least until the next Python version. Using private APIs is risky, but I couldn't find a better way.

src/functional_tests/base.py (ch25l007)

```
import os
import time
from datetime import datetime
from pathlib import Path
[...]
MAX_WAIT = 5

SCREEN_DUMP_LOCATION = Path(__file__).absolute().parent / "screendumps"
[...]
class FunctionalTest(StaticLiveServerTestCase):
    def setUp(self):
        [...]

    def tearDown(self):
        if self._test_has_failed():
            if not SCREEN_DUMP_LOCATION.exists():
                SCREEN_DUMP_LOCATION.mkdir(parents=True)
            self.take_screenshot()
            self.dump_html()
        self.browser.quit()
        super().tearDown()

    def _test_has_failed(self):
        # slightly obscure but couldn't find a better way!
        return self._outcome.result.failures or self._outcome.result.errors
```

We first create a directory for our screenshots if necessary, and then we take our screenshot and dump the HTML. Let's see how those will work:

src/functional_tests/base.py (ch25l008)

```
    def take_screenshot(self):
        path = SCREEN_DUMP_LOCATION / self._get_filename("png")
        print("screenshotting to", path)
        self.browser.get_screenshot_as_file(str(path))

    def dump_html(self):
        path = SCREEN_DUMP_LOCATION / self._get_filename("html")
        print("dumping page HTML to", path)
        path.write_text(self.browser.page_source)
```

And finally, here's a way of generating a unique filename identifier, which includes the name of the test and its class, as well as a timestamp:

src/functional_tests/base.py (ch25l009)

```
    def _get_filename(self, extension):
        timestamp = datetime.now().isoformat().replace(":", ".")[:19]
        return (
            f"{self.__class__.__name__}.{self._testMethodName}-{timestamp}.{extension}"
        )
```

You can test this first locally by deliberately breaking one of the tests—with a `self.fail()` half-way through, for example—and you'll see something like this:

```
$ ./src/manage.py test functional_tests.test_my_lists
[...]
Fscreenshotting to ...goat-book/src/functional_tests/screendumps/MyListsTest.te
st_logged_in_users_lists_are_saved_as_my_lists-[...]
dumping page HTML to ...goat-book/src/functional_tests/screendumps/MyListsTest.
test_logged_in_users_lists_are_saved_as_my_lists-[...]
Fscreenshotting to ...goat-book/src/functional_tests/screendumps/MyListsTest.te
st_logged_in_users_lists_are_saved_as_my_lists-2025-02-18T11.29.00.png
dumping page HTML to ...goat-book/src/functional_tests/screendumps/MyListsTest.
test_logged_in_users_lists_are_saved_as_my_lists-2025-02-18T11.29.00.html
```

Why not try and open one of those files up? It's kind of satisfying.

Saving Build Outputs (or Debug Files) as Artifacts

We also need to tell GitLab to "save" these files, for us to be able to actually look at them. Those are called *artifacts*:

.gitlab-ci.yml (ch25l012)

```
test:
  [...]

  script:
    [...]

  artifacts: ❶
    when: always  ❷
    paths: ❶
      - src/functional_tests/screendumps/
```

❶ `artifacts` is the name of the key, and the `paths` argument is fairly self-explanatory. You can use wildcards here—more info in the GitLab docs.

❷ One thing the docs *didn't* make obvious is that you need `when: always`, because otherwise it won't save artifacts for failed jobs. That was annoyingly hard to figure out!

In any case, that should work. If you commit the code and then push it back to GitLab, we should be able to see a new build job:

```
$ echo "src/functional_tests/screendumps" >> .gitignore
$ git commit -am "add screenshot on failure to FT runner"
$ git push
```

In its output, we'll see the screenshots and HTML dumps being saved:

```
screendumps/LoginTest.test_can_get_email_link_to_log_in-window0-2014-01-22T17.45.12.html
Fscreenshotting to /builds/hjwp/book-example/src/functional_tests/screendumps/
NewVisitorTest.test_can_start_a_todo_list-2025-02-17T17.51.01.png
dumping page HTML to /builds/hjwp/book-example/src/functional_tests/screendumps/
NewVisitorTest.test_can_start_a_todo_list-2025-02-17T17.51.01.html
Not Found: /favicon.ico
.screenshotting to /builds/hjwp/book-example/src/functional_tests/screendumps/
NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls-2025-02-17T17.
51.06.png
dumping page HTML to /builds/hjwp/book-example/src/functional_tests/screendumps/
NewVisitorTest.test_multiple_users_can_start_lists_at_different_urls-2025-02-17T17.51.
06.html
======================================================================
FAIL: test_can_start_a_todo_list (functional_tests.test_simple_list_creation.NewVisitorTest.
test_can_start_a_todo_list)
[...]
```

And to the right, some new UI options appear to Browse the artifacts, as in Figure 25-4.

Figure 25-4. Artifacts appear on the right of the build job

And if you navigate through, you'll see something like Figure 25-5.

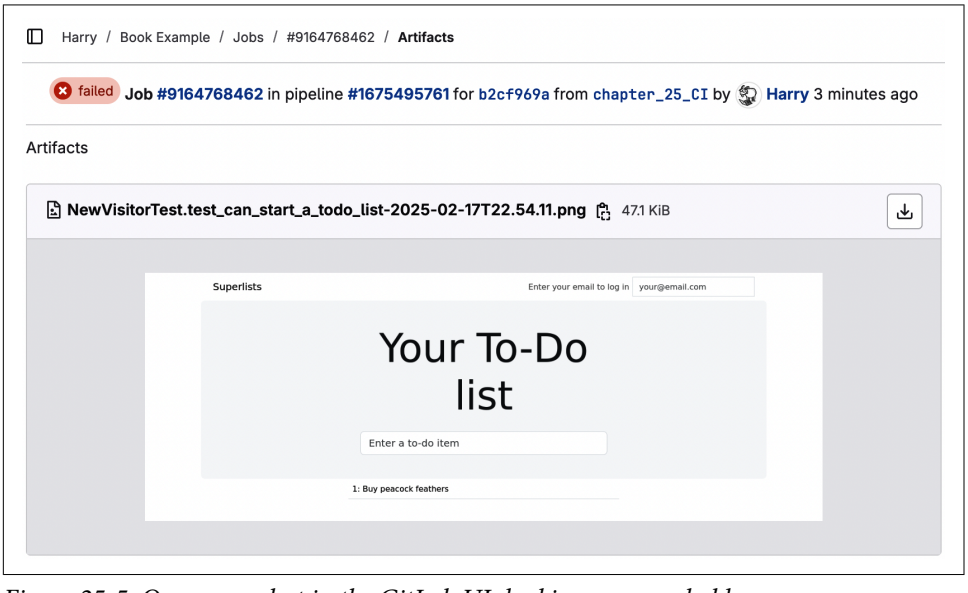


Figure 25-5. Our screenshot in the GitLab UI, looking unremarkable

If in Doubt, Try Bumping the Timeout!

Your build might be clear, but mine was still failing, and those screenshots didn't offer any obvious clues. Hmm. Well, when in doubt, bump the timeout—as the old adage goes:

src/functional_tests/base.py

```
MAX_WAIT = 10
```

Then we can rerun the build by pushing, and confirm it now works.

A Successful Python Test Run

At this point, we should get a working pipeline (see Figure 25-6).

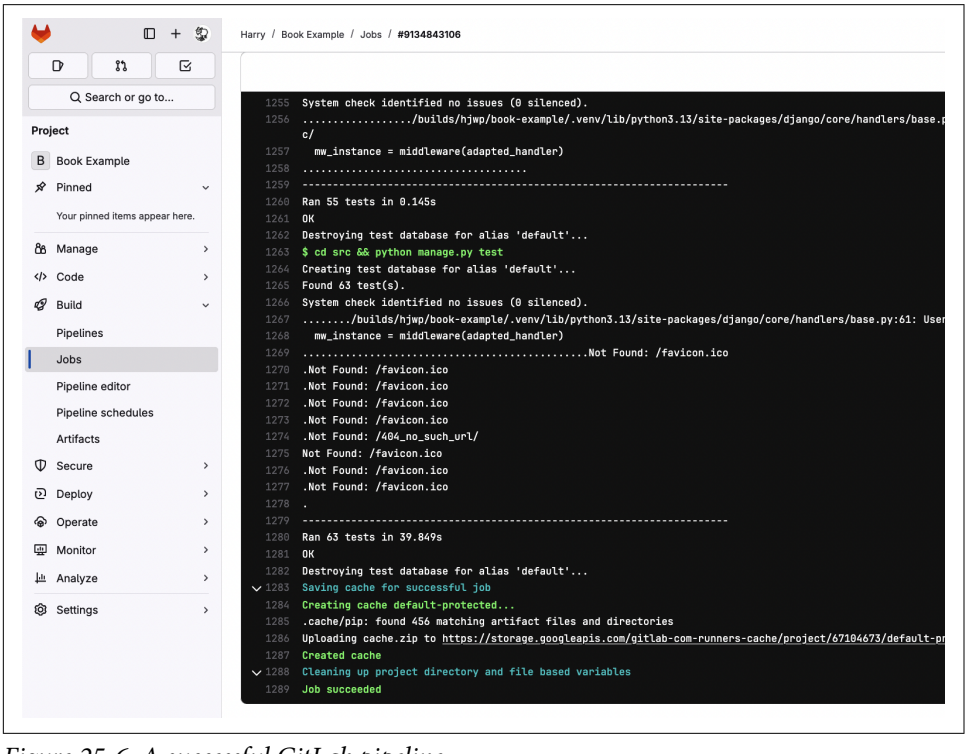


Figure 25-6. A successful GitLab pipeline

Running Our JavaScript Tests in CI

There's a set of tests we almost forgot—the JavaScript tests. Currently our "test runner" is an actual web browser. To get them running in CI, we need a command-line test runner.

Our JavaScript tests currently test the interaction between our code and the Bootstrap framework/CSS, so we still need a real browser to be able to make our visibility checks work.

Thankfully, the Jasmine docs point us straight towards the kind of tool we need: Jasmine browser runner.

Installing Node.js

It's time to stop pretending we're not in the JavaScript game. We're doing web development; that means we do JavaScript; that means we're going to end up with Node.js on our computers. It's just the way it has to be.

Follow the instructions on the Node.js home page. It should guide you through installing the "node version manager" (nvm), and then to getting the latest version of node:

```
$ nvm install --lts
Installing Node v22.17.0 (arm64)
[...]
$ node -v
v22.17.0
```

Installing and Configuring the Jasmine Browser Runner

The docs suggest we install it like this, and then run the `init` command to generate a default config file:

```
$ cd src/lists/static

$ npm install --save-dev jasmine-browser-runner jasmine-core
[...]
added 151 packages in 4s

$ cat package.json  # this is the equivalent of requirements.txt
{
  "devDependencies": {
    "jasmine-browser-runner": "^3.0.0",
    "jasmine-core": "^5.6.0"
  }
}

$ ls node_modules/
# will show several dozen directories

$ npx jasmine-browser-runner init
Wrote configuration to spec/support/jasmine-browser.mjs.
```

Well, we now have about a million files in *node_modules/* (which is JavaScript's version of a virtualenv, essentially), and we also have a new config file in *spec/support/jasmine-browser.mjs*. That's not the ideal place, because we've said our tests live in a folder called *tests*. So, let's move the config file in there:

```
$ mv spec/support/jasmine-browser.mjs tests/jasmine-browser-runner.config.mjs
$ rm -rf spec
```

Then let's edit it slightly, to specify a few things correctly:

src/lists/static/tests/jasmine-browser-runner.config.mjs (ch25l013)

```
export default {
  srcDir: ".",  ❶
  srcFiles: [
    "*.js"
  ],
  specDir: "tests",  ❷
  specFiles: [
    "**/*[sS]pec.js"
  ],
  helpers: [
    "helpers/**/*.js"
  ],
  env: {
    stopSpecOnExpectationFailure: false,
    stopOnSpecFailure: false,
    random: true,
    forbidDuplicateNames: true
  },
  listenAddress: "localhost",
  hostname: "localhost",
  browser: {
    name: "headlessFirefox"  ❸
  }
};
```

❶ Our source files are in the current directory, *src/lists/static*—i.e., *lists.js*.

❷ Our spec files are in *tests/*.

❸ And here we say we want to use the headless version of Firefox. (We could have done this by setting `MOZ_HEADLESS` at the command line again, but this saves us from having to remember.)

Let's try running it now. We use the `--config` option to pass it to the now non-standard path to the config file:

```
$ npx jasmine-browser-runner runSpecs \
  --config=tests/jasmine-browser-runner.config.mjs
Jasmine server is running here: http://localhost:62811
Jasmine tests are here:         ...goat-book/src/lists/static/tests
Source files are here:          ...goat-book/src/lists/static
Running tests in the browser...
Randomized with seed 17843
Started
.F.

Failures:
1) Superlists tests error message should be hidden on input
  Message:
    Expected true to be false.
  Stack:
    <Jasmine>
    @http://localhost:62811/spec/Spec.js:46:40
    <Jasmine>

3 specs, 1 failure
Finished in 0.014 seconds
Randomized with seed 17843 (jasmine-browser-runner runSpecs --seed=17843)
```

Could be worse! One failure out of three specs. Unfortunately, it's the most important test:

src/lists/static/tests/Spec.js

```
it("should hide error message on input", () => {
  initialize(inputSelector);
  textInput.dispatchEvent(new InputEvent("input"));

  expect(errorMsg.checkVisibility()).toBe(false);
});
```

Ah yes, if you remember, I said that the main reason we need to use a browser-based test runner is because our visibility checks depend on the Bootstrap CSS framework.

In the HTML spec runner we've configured so far, we load Bootstrap using a `<link>` tag:

src/lists/static/tests/SpecRunner.html

```
<!-- Bootstrap CSS -->
<link href="../bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

And here's how we load it for `jasmine-browser-runner`:

src/lists/static/tests/jasmine-browser-runner.config.mjs (ch25l014)

```
export default {
  srcDir: ".",
  srcFiles: [
    "*.js"
  ],
  specDir: "tests",
  specFiles: [
    "**/*[sS]pec.js"
  ],
  cssFiles: [  ❶
    "bootstrap/css/bootstrap.min.css"  ❶
  ],
  helpers: [
    "helpers/**/*.js"
  ],
```

❶ The `cssFiles` key is how you tell the runner to load, er, some CSS. I found that out in the docs.

Let's give that a go…

```
$ npx jasmine-browser-runner runSpecs \
  --config=tests/jasmine-browser-runner.config.mjs
Jasmine server is running here: http://localhost:62901
Jasmine tests are here:         .../goat-book/src/lists/static/tests
Source files are here:          .../goat-book/src/lists/static
Running tests in the browser...
Randomized with seed 06504
Started
...


3 specs, 0 failures
Finished in 0.016 seconds
Randomized with seed 06504 (jasmine-browser-runner runSpecs --seed=06504)
```

Hooray! That works locally—let's get it into CI:

```
$ cd -  # go back to the project root
# add the package.json, which saves our node depenencies
$ git add src/lists/static/package.json src/lists/static/package-lock.json
# ignore the node_modules/ directory
$ echo "node_modules/" >> .gitignore
# and our config file
$ git add src/lists/static/tests/jasmine-browser-runner.config.mjs
$ git add .gitignore
$ git commit -m "config for node + jasmine-browser-runner for JS tests"
```

Adding a Build Step for JavaScript

We now want two different build steps, so let's rename `test` to `test-python` and move all its specific bits like `variables` and `before_script` inside it, and then create a separate step called `test-js`, with a similar structure:

.gitlab-ci.yml (ch25l018)

```
test-python:
  # Use the same image as our Dockerfile
  image: python:slim  ❶

  variables:  ❶
    # Put pip-cache in home folder so we can use gitlab cache
    PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"
    # Make Firefox run headless.
    MOZ_HEADLESS: "1"

  cache:  ❶
    paths:
      - .cache/pip

  # "setUp" phase, before the main build
  before_script:  ❶
    - python --version ; pip --version  # For debugging
    - pip install virtualenv
    - virtualenv .venv
    - source .venv/bin/activate

  script:
    - pip install -r requirements.txt
    # unit tests
    - python src/manage.py test lists accounts
    # (if those pass) all tests, incl. functional.
    - apt update -y && apt install -y firefox-esr
    - pip install selenium
    - cd src && python manage.py test

  artifacts:
    when: always
    paths:
      - src/functional_tests/screendumps/

test-js:  ❷
  image: node:slim
  script:
    - apt update -y && apt install -y firefox-esr  ❸
    - cd src/lists/static
    - npm install  ❹
    - npx jasmine-browser-runner runSpecs
      --config=tests/jasmine-browser-runner.config.mjs  ❺
```

❶ `image`, `variables`, `cache`, and `before_script` all move out of the top level and into the `test-python` step, as they're all specific to this step only now.

❷ Here's our new step, `test-js`.

❸ We install Firefox into the node image, just like we do for the Python one.

❹ We don't need to specify *what* to `npm install`, because that's all in the *package-lock.json* file.

❺ And here's our command to run the tests.

And slap me over the head with a wet fish if that doesn't pass on the first go! See Figure 25-7 for a successful pipeline run.

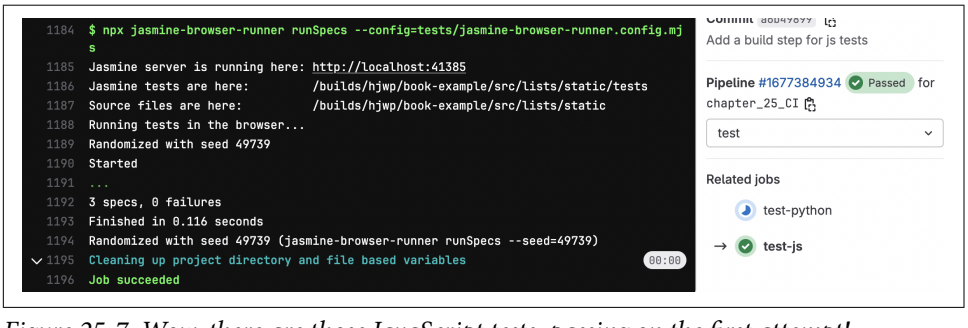


Figure 25-7. Wow, there are those JavaScript tests, passing on the first attempt!

Tests Now Pass

And there we are! A complete CI build featuring all of our tests! See Figure 25-8.

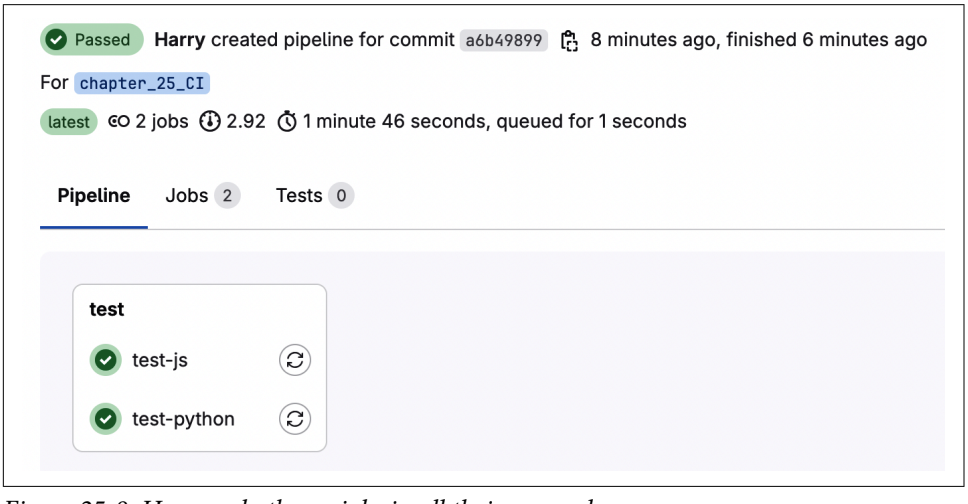


Figure 25-8. Here are both our jobs in all their green glory

Nice to know that, no matter how lazy I get about running the full test suite on my own machine, the CI server will catch me. Another one of the Testing Goat's agents in cyberspace, watching over us…

Alternatives: Woodpecker and Forgejo

I want to give a shout out to Woodpecker CI and Forgejo, two of the newer self-hosted CI options. And while I'm at it, to Jenkins, which did a great job for the first and second editions, and still does for many people.

If you want true independence from overly commercial interests, then self-hosted is the way to go. You'll need your own server for both of these.

I tried both, and managed to get them working within an hour or two. Their documentation is good.

If you do decide to give them a go, I'd say, be a bit cautious about security options. For example, you might decide you don't want any old person from the internet to be able to sign up for an account on your server:

```
DISABLE_REGISTRATION: true
```

But more power to you for giving it a go!

Some Things We Didn't Cover

CI is a big topic and, inevitably, I couldn't cover everything. Here's a few pointers to things you might want to learn about.

Defining a Docker Image for CI

We spent quite a bit of time debugging—for example, the unhelpful messages when Firefox wasn't installed. Just as we did when preparing our deployment, it's a big help having an environment that you can run on your local machine that's as close as possible to what you have remotely; that's why we chose to use a Docker image.

In CI, our tests also run a Docker image (`python:slim` and `node:slim`), so one common pattern is to define a Docker image within your repo that you'll use for CI. Ideally, it should also be as similar as possible to the one you use in production! A typical solution here is to use multistage Docker builds—with a base stage, a prod stage, and a dev/CI stage. In our case, the last stage would have Firefox, Selenium, and other test-only dependencies in it, which we don't need for prod.

You can then run your tests locally inside the same Docker image that's used in CI.

Reproducibility is one of the key attributes we're aiming for. The more your project grows in complexity, the more it's worth investing in minimising the differences between local dev, CI, and prod.

Caching

We touched on the use of caches in CI for the `pip` download cache, but as CI pipelines grow in maturity, you'll find you can make more and more use of caching. For example, it might be a good idea to cache your *node_modules/* directory.

It's a topic for another time, but this is yet another way of trying to speed up the feedback cycle.

Automated Deployment, aka Continuous Delivery (CD)

The natural next step is to finish our journey into automation, and set up a pipeline that will deploy our code all the way to production, each time we push code…as long as the tests pass!

I work through an example of how to do that in the Online Appendix: Continuous Deployment (CD). If you're feeling inspired, I'd encourage you to take a look.

Now, onto our last chapter of coding, everyone!

Best Practices for CI (Including Selenium Tips)

Set up CI as soon as possible for your project.
: As soon as your functional tests take more than a few seconds to run, you'll find yourself avoiding running them. Give this job to a CI server, to make sure that all your tests are being run somewhere.

Optimise for fast feedback.
: CI feedback loops can be frustratingly slow. Optimising things to get results quicker is worth the effort. Run your fastest tests first, and use caches to try to minimise time spent on, for example, dependency installation.

Set up screenshots and HTML dumps for failures.
: Debugging test failures is easier if you can see what the page looked like when the failure occurred. This is particularly useful for debugging CI failures, but it's also very useful for tests that you run locally.

Be prepared to bump your timeouts.
: A CI server may not be as speedy as your laptop—especially if it's under load, running multiple tests at the same time. Be prepared to be even more generous with your timeouts, in order to minimise the chance of random failures.

Take the next step, CD (continuous deployment).
: Once we're running tests automatically, we can take the next step, which is to automate our deployments (when the tests pass). See the Online Appendix: Continuous Deployment (CD).

CHAPTER 26

The Token Social Bit, the Page Pattern, and an Exercise for the Reader

Are jokes about how "everything has to be social now" slightly old hat? Yes, Harry; they were old hat 10 years ago when you started writing this book, and they're positively prehistoric now. *Irregardless*, let's say lists are often better shared. We should allow our users to collaborate on their lists with other users.

Along the way, we'll improve our FTs by starting to implement something called the "page object pattern". Then, rather than showing you explicitly what to do, I'm going to let you write your unit tests and application code by yourself. Don't worry; you won't be totally on your own! I'll give an outline of the steps to take, as well as some hints and tips.

But still—if you haven't already, this is the chapter where you get a chance to spread your wings. Enjoy!

An FT with Multiple Users, and addCleanup

Let's get started—we'll need two users for this FT:

src/functional_tests/test_sharing.py (ch26l001)

```
from selenium import webdriver
from selenium.webdriver.common.by import By

from .base import FunctionalTest


def quit_if_possible(browser):
    try:
        browser.quit()
    except:
        pass


class SharingTest(FunctionalTest):
    def test_can_share_a_list_with_another_user(self):
        # Edith is a logged-in user
        self.create_pre_authenticated_session("edith@example.com")
        edith_browser = self.browser
        self.addCleanup(lambda: quit_if_possible(edith_browser))

        # Her friend Onesiphorus is also hanging out on the lists site
        oni_browser = webdriver.Firefox()
        self.addCleanup(lambda: quit_if_possible(oni_browser))
        self.browser = oni_browser
        self.create_pre_authenticated_session("onesiphorus@example.com")

        # Edith goes to the home page and starts a list
        self.browser = edith_browser
        self.browser.get(self.live_server_url)
        self.add_list_item("Get help")

        # She notices a "Share this list" option
        share_box = self.browser.find_element(By.CSS_SELECTOR, 'input[name="sharee"]')
        self.assertEqual(
            share_box.get_attribute("placeholder"),
            "your-friend@example.com",
        )
```

The interesting feature to note about this section is the `addCleanup` function, whose documentation you can find online. It can be used as an alternative to the `tearDown` function as a way of cleaning up resources used during the test. It's most useful when the resource is only allocated halfway through a test, so you don't have to spend time in `tearDown` with a bunch of conditional logic designed to clean up resources that may or may not have been used by the point the test failed.

`addCleanup` is run after `tearDown`, which is why we need that `try/except` formulation for `quit_if_possible`. By the time the test ends, the browser assigned to `self.browser`—whether it was `edith_browser` or `oni_browser`—will already have been quit by `tearDown()`.

We'll also need to move `create_pre_authenticated_session` from *test_my_lists.py* into *base.py*, so we can use it in more than one test.

OK, let's see if that all works:

```
$ python src/manage.py test functional_tests.test_sharing
[...]
Traceback (most recent call last):
  File "...goat-book/src/functional_tests/test_sharing.py", line 33, in
test_can_share_a_list_with_another_user
[...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: input[name="sharee"]; [...]
```

Great! It seems to have made it through creating the two user sessions, and it gets onto an expected failure—there is no input for an email address of a person to share a list with on the page.

Let's do a commit at this point, because we've got at least a placeholder for our FT, we've got a useful modification of the `create_pre_authenticated_session` function, and we're about to embark on a bit of an FT refactor:

```
$ git add src/functional_tests
$ git commit -m "New FT for sharing, move session creation stuff to base"
```

The Page Pattern

Before we go any further, I want to show an alternative method for reducing duplication in your FTs, called "page objects".

We've already built several helper methods for our FTs—including `add_list_item`, which we've used here—but if we just keep adding more and more, it's going to get very crowded. I've worked on a base FT class that was over 1,500 lines long, and that got pretty unwieldy.

Page objects are an alternative that encourage us to store all the information and helper methods about the different types of pages on our site in a single place. Let's see how that might look for our site, starting with a class to represent any lists page:

src/functional_tests/list_page.py

```
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

from .base import wait


class ListPage:
    def __init__(self, test):
        self.test = test  ❶

    def get_table_rows(self):  ❸
        return self.test.browser.find_elements(By.CSS_SELECTOR, "#id_list_table tr")

    @wait
    def wait_for_row_in_list_table(self, item_text, item_number):  ❷
        expected_row_text = f"{item_number}: {item_text}"
        rows = self.get_table_rows()
        self.test.assertIn(expected_row_text, [row.text for row in rows])

    def get_item_input_box(self):  ❷
        return self.test.browser.find_element(By.ID, "id_text")

    def add_list_item(self, item_text):  ❷
        new_item_no = len(self.get_table_rows()) + 1
        self.get_item_input_box().send_keys(item_text)
        self.get_item_input_box().send_keys(Keys.ENTER)
        self.wait_for_row_in_list_table(item_text, new_item_no)
        return self  ❹
```

❶ It's initialised with an object that represents the current test. That gives us the ability to make assertions, access the browser instance via `self.test.browser`, and use the `self.test.wait_for` function.

❷ I've copied across some of the existing helper methods from *base.py*, but I've tweaked them slightly…

❸ For example, this new method is used in the new versions of the old helper methods.

❹ Returning `self` is just a convenience. It enables method chaining, which we'll see in action immediately.

Let's see how to use it in our test:

src/functional_tests/test_sharing.py (ch26l004)

```
from .list_page import ListPage
[...]

        # Edith goes to the home page and starts a list
        self.browser = edith_browser
        self.browser.get(self.live_server_url)
        list_page = ListPage(self).add_list_item("Get help")
```

Let's continue rewriting our test, using the page object whenever we want to access elements from the lists page:

src/functional_tests/test_sharing.py (ch26l008)

```
        # She notices a "Share this list" option
        share_box = list_page.get_share_box()
        self.assertEqual(
            share_box.get_attribute("placeholder"),
            "your-friend@example.com",
        )

        # She shares her list.
        # The page updates to say that it's shared with Onesiphorus:
        list_page.share_list_with("onesiphorus@example.com")
```

We add the following three functions to our `ListPage`:

src/functional_tests/list_page.py (ch26l009)

```
    def get_share_box(self):
        return self.test.browser.find_element(
            By.CSS_SELECTOR,
            'input[name="sharee"]',
        )

    def get_shared_with_list(self):
        return self.test.browser.find_elements(
            By.CSS_SELECTOR,
            ".list-sharee",
        )

    def share_list_with(self, email):
        self.get_share_box().send_keys(email)
        self.get_share_box().send_keys(Keys.ENTER)
        self.test.wait_for(
            lambda: self.test.assertIn(
                email, [item.text for item in self.get_shared_with_list()]
            )
        )
```

The idea behind the page pattern is that it should capture all the information about a particular page in your site. That way, if you later want to go and make changes to that page—even just simple tweaks to its HTML layout—you'll have a single place to adjust your functional tests, rather than having to dig through dozens of FTs.

The next step would be to pursue the FT refactor through our other tests. I'm not going to show that here, but it's something you could do for practice, to get a feel for what the trade-offs are like between "don't repeat yourself" (DRY) and test readability…

Extend the FT to a Second User, and the "My Lists" Page

Let's spec out just a little more detail of what we want our sharing user story to be. Edith has seen on her list page that the list is now "shared with" Onesiphorus, and then we can have Onesiphorus log in and see the list on his "My lists" page—maybe in a section called "lists shared with me":

src/functional_tests/test_sharing.py (ch26l010)

```
from .my_lists_page import MyListsPage
[...]

        list_page.share_list_with("onesiphorus@example.com")

        # Onesiphorus now goes to the lists page with his browser
        self.browser = oni_browser
        MyListsPage(self).go_to_my_lists_page("onesiphorus@example.com")

        # He sees Edith's list in there!
        self.browser.find_element(By.LINK_TEXT, "Get help").click()
```

That means another function in our `MyListsPage` class:

src/functional_tests/my_lists_page.py (ch26l011)

```
from selenium.webdriver.common.by import By


class MyListsPage:
    def __init__(self, test):
        self.test = test

    def go_to_my_lists_page(self, email):
        self.test.browser.get(self.test.live_server_url)
        self.test.browser.find_element(By.LINK_TEXT, "My lists").click()
        self.test.wait_for(
            lambda: self.test.assertIn(
                email,
                self.test.browser.find_element(By.TAG_NAME, "h1").text,
            )
        )
        return self
```

Once again, this is a function that would be good to carry across into *test_my_lists.py*, along with maybe a `MyListsPage` object.

In the meantime, Onesiphorus can also add things to the list:

src/functional_tests/test_sharing.py (ch26l012)

```
    # On the list page, Onesiphorus can see says that it's Edith's list
    self.wait_for(
        lambda: self.assertEqual(list_page.get_list_owner(), "edith@example.com")
    )

    # He adds an item to the list
    list_page.add_list_item("Hi Edith!")

    # When Edith refreshes the page, she sees Onesiphorus's addition
    self.browser = edith_browser
    self.browser.refresh()
    list_page.wait_for_row_in_list_table("Hi Edith!", 2)
```

That's another addition to our `ListPage` object:

src/functional_tests/list_page.py (ch26l013)

```
class ListPage:
    [...]

    def get_list_owner(self):
        return self.test.browser.find_element(By.ID, "id_list_owner").text
```

It's long past time to run the FT and check if all of this works!

```
$ python src/manage.py test functional_tests.test_sharing
[...]
  File "...goat-book/src/functional_tests/test_sharing.py", line 35, in
test_can_share_a_list_with_another_user
    share_box = list_page.get_share_box()
    [...]
    return self.test.browser.find_element(
           ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^
        By.CSS_SELECTOR,
        ^^^^^^^^^^^^^^^^
        'input[name="sharee"]',
        ^^^^^^^^^^^^^^^^^^^^^^^
    [...]
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate
element: input[name="sharee"]; [...]
```

That's the expected failure; we don't have an input for email addresses of people to share with. Let's do a commit:

```
$ git add src/functional_tests
$ git commit -m "Create Page objects for list pages, use in sharing FT"
```

An Exercise for the Reader

> I probably didn't *really* understand what I was doing until after having completed the "exercise for the reader" in the page pattern chapter.
>
> —Iain H. (reader)

There's nothing that cements learning like taking the training wheels off, and getting something working on your own, so I hope you'll give this a go.

By this point in the book, you should have all the elements you need to test-drive this new feature, from the outside in. The FT is there to guide you, and this feature should take you down into both the views and the models layers. So, give it a go!

Step-by-Step Guide

If you'd like a bit more help, here's an outline of the steps you could take:

1. You'll need a new section in *list.html*, initially with just a form containing an input box for an email address. That should get the FT one step further.
2. Next, you'll need a view for the form to submit to. Start by defining the URL in the template—maybe something like *lists/<list_id>/share*.
3. Then, you'll have your first unit test. It can be just enough to get a placeholder view in. You want the view to respond to POST requests and respond with a

redirect back to the list page. The test could be called something like `ShareListTest.test_post_redirects_to_lists_page`.

4. You build out your placeholder view, as just a two-liner that finds a list and redirects to it.
5. You can then write a new unit test that creates a user and a list, does a POST with their email address, and checks that the user is added to `mylist.shared_with.all()` (a similar ORM usage to "My lists"). That `shared_with` attribute won't exist yet; you're going outside-in.
6. So, before you can get this test to pass, you'll have to move down to the model layer. The next test, in *test_models.py*, can check that a list has a `shared_with.add()` method that works with a user's email address, and that `shared_with.all()` subsequently includes that user.
7. You'll then need a `ManyToManyField`. You'll probably see an error message about a clashing `related_name`, which you'll find a solution for if you look around the Django docs.
8. It will need a database migration.
9. That should get the model tests passing. Pop back up to fix the view test.
10. You may find that the redirect view test fails, because it's not sending a valid POST request. You can either choose to ignore invalid inputs, or adjust the test to send a valid POST.
11. Then, head back up to the template level; on the "My lists" page, you'll want a `<ul>` with a `for` loop of the lists shared with the user. On the lists page, you also want to show who the list is shared with, and mention who the list owner is. Look back at the FT for the correct classes and IDs to use. You could have brief unit tests for each of these if you like, as well.
12. You might find that spinning up the site with `runserver` helps you iron out any bugs and fine-tune the layout and aesthetics. If you use a private browser session, you'll be able to log multiple users in.

By the end, you might end up with something that looks like Figure 26-1.

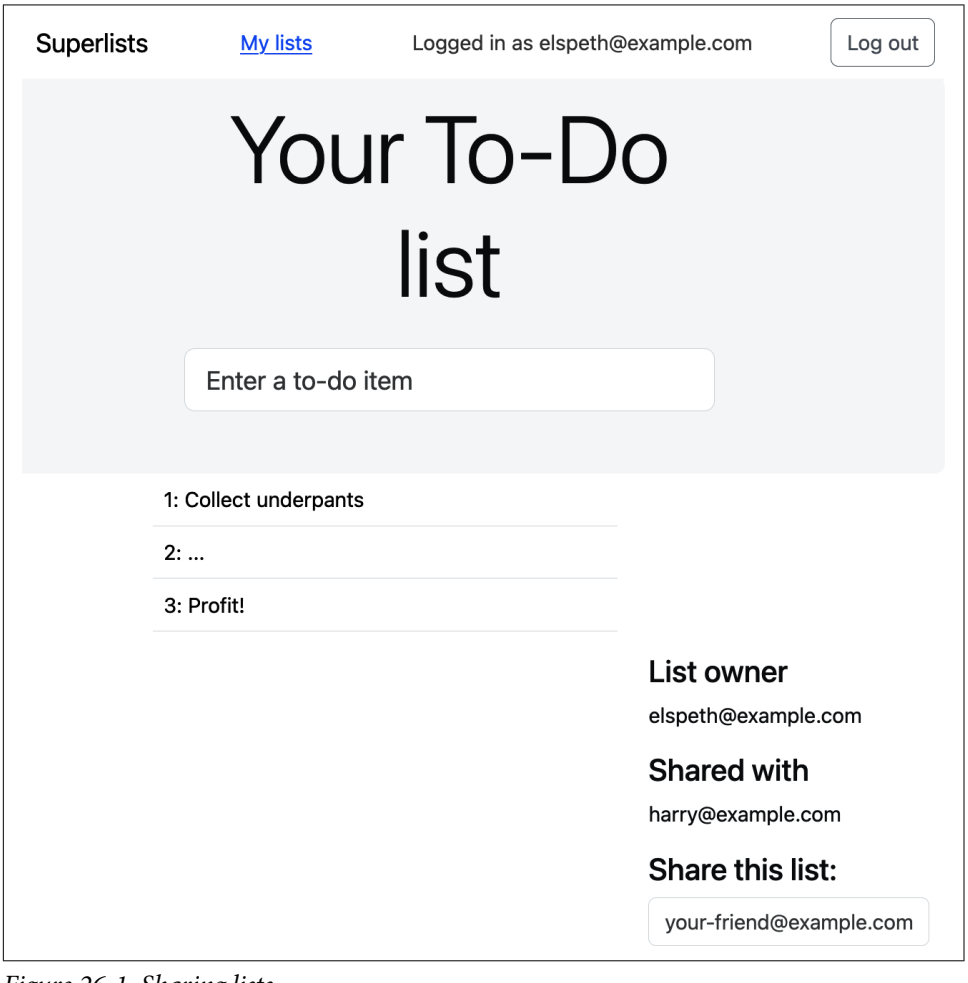


Figure 26-1. Sharing lists

The Page Pattern, and the Real Exercise for the Reader

Applying DRY to your functional tests
: Once your FT suite starts to grow, you'll find different tests using similar parts of the UI. Try to avoid having constants—like the HTML IDs or classes of particular UI elements—duplicated across your FTs.

The page pattern
: Moving helper methods into a base `FunctionalTest` class can become unwieldy. Consider using individual page objects to hold all the logic for dealing with particular parts of your site.

An exercise for the reader
: I hope you've actually tried this out! Try to follow the outside-in method, and occasionally try things out manually if you get stuck. The real exercise for the reader, of course, is to apply TDD to your next project. I hope you'll enjoy it!

In the next chapter, we'll wrap up with a discussion of the trade-offs in testing, and some of the considerations involved in choosing which kinds of tests to use, and when.

CHAPTER 27

Fast Tests, Slow Tests, and Hot Lava

> The database is hot lava!
>
> —Casey Kinsey

We've come to the end of the book, and the end of our journey with this to-do app and its tests. Let's recap our test structure so far:

- We have a suite of functional tests that use Selenium to test that the whole app really works. On several occasions, the FTs have saved us from shipping broken code—whether it was broken CSS, a broken database due to filesystem permissions, or broken email integration.
- And we have a suite of unit tests that use Django test client, enabling us to test-drive our code for models, forms, views, URLs, and even (to some extent) templates. They've enabled us to build the app incrementally, to refactor with confidence, and they've supported a fast unit-test/code cycle.
- We've also spent a good bit of time on our infrastructure, packaging up our app with Docker for ease of deployment, and we've set up a CI pipeline to run our tests automatically on push.

However, being a simple app that could fit in a book, there are inevitably some limitations and simplifications in our approach. In this chapter, I'd like to talk about how to carry your testing principles forward, as you move into larger, more complex applications in the real world.

Let's find out why someone might say that the database is hot lava!

Why Do We Test? Our Desiderata for Effective Tests

At *testdesiderata.com*, Kent Beck and Kelly Sutton outline several desiderata (desirable characteristics) for tests:

- **Isolated**: Tests should return the same results regardless of the order in which they are run.
- **Composable**: We should be able to test different dimensions of variability separately and combine the results.
- **Deterministic**: If nothing changes, the test results shouldn't change.
- **Fast**: Tests should run quickly.
- **Writable**: Tests should be cheap to write, relative to the cost of the code being tested.
- **Readable**: Tests should be comprehensible for readers, invoking the motivation for writing the particular test.
- **Behavioural**: Tests should be sensitive to changes in the behaviour of the code under test. If the behaviour changes, the test result should change.
- **Structure-agnostic**: Tests should not change their results if the structure of the code changes.
- **Automated**: Tests should run without human intervention.
- **Specific**: If a test fails, the cause of the failure should be obvious.
- **Predictive**: If the tests all pass, then the code under test should be suitable for production.
- **Inspiring**: Passing the tests should inspire confidence.

We've talked about almost all of these desiderata in the book: we talked about *isolation* when we switched to using the Django test runner. We talked about *composability* when discussing the car factory example in Chapter 21. We talked about tests being *readable* when we talked about the given-when-then structure and when implementing helper methods in our FTs. We talked about testing *behaviour* rather than implementation at several points, including in the mocking chapters. We talked about *structure* in the forms chapters, when we showed that the higher-level views tests enabled us to refactor more freely than the lower-level forms tests. We've talked about splitting up our tests to have fewer assertions to make them more *specific*. We talked about *determinism* when discussing flaky tests and the use of `wait_for()` in our FTs, for example, as well as in the production debugging chapter.

And in this chapter, we're going to talk primarily about *speed*, and about what makes tests *inspiring*.

But first, it's worth taking a step back from the preceding list, and asking: "What do we want from our tests?"

Confidence and Correctness (Preventing Regression)

A fundamental part of programming is that, now and again, you need to check whether "it works". Automated testing is the solution to the problem that checking things manually can quickly become tedious and be unreliable. We want our tests to tell us that our code works—both at the low level of individual functions or classes, and at the higher level of "does it all hang together?"

A Productive Workflow

Our tests need to be fast enough to write, but more importantly, fast to run. We want to get into a smooth, productive workflow, and try to enter that holy credo of programmers—the "flow state". Beyond that, we want our tests to take some of the stress out of programming, encouraging us to work in small increments, with frequent bursts of dopamine from seeing green tests.

Driving Better Design

And our tests should help us to write *better* code: first, by enabling fearless refactoring and, second, by giving us feedback on the design of our code. Writing the tests first lets us think about our API from the outside in, before we write it—and we've seen that. But in this chapter, we'll also talk about the potential for tests to give you feedback on your design in more subtle ways. As we'll see, designing code to be more testable often leads to code that has clearly identified dependencies, and is more modular and more decoupled.

As we continuously think about what kinds of tests to write, we are trying to achieve the optimum balance of these different desiderata.

Were Our Unit Tests Integration Tests All Along? What Is That Warm Glow Coming from the Database?

Almost all of the "unit" tests in the book perhaps should have been called *integration* tests, because they all rely on Django's test runner, which gives us a real database to talk to. Many also use the Django test client, which does a lot of magic with the middleware layers that sit between requests. The end result is that our tests are heavily integrated with both the database and Django itself.

We've Been in the "Sweet Spot"

Now, actually, this has been a pretty good thing for us so far. We're very much in the "sweet spot" of Django's testing tools. Our unit tests have been fast enough to enable a smooth workflow, and they've given us a strong reassurance that our application really works—from the models all the way up to the templates. By allying them with a small-ish suite of functional tests, we've got a lot of confidence in our code. And we've been able to use them to get at least a bit of feedback on our design, and to enable lots of refactoring.

What Is a "True" Unit Test? Does it Matter?

But people will often tell you that a "true" unit test should be more isolated. It's meant to test a single "unit" of software, and your database "should" be outside of that. Why do they say that (other than for the smugness they get from should-ing us)?

As you can tell, I think the argument from *definitions* is a bit of a red herring. But you might hear instead, "the database is hot lava!"—as Casey Kinsey put it in a memorable DjangoCon talk. There is real feeling and real experience behind these comments. What are people getting at?

Integration and Functional Tests Get Slower Over Time

The problem is that, as your application and codebase grow, involving the database in every single test starts to carry an unacceptable cost—in terms of execution speed. Casey's company, for example, was struggling with test suites that took several hours.

At PythonAnywhere, our functional test suite didn't just rely on the database; it would spin up a full test cluster of six virtual machines. A full run used to take at least 12 hours, and we'd have to wait overnight for our results. That was one of the least productive parts of an otherwise extraordinary workflow.

At Kraken, the full test suite does only take about 45 minutes, which is not bad for nearly 10 million lines of code, but that's only thanks to a quite frankly ridiculous level of parallelisation and associated expenditure on CI. We're now spending a lot of effort on trying to move more of our unit tests to being "true" unit tests.

The problem is that these things don't scale linearly. The more database tables you have, the more relationships between them, and that starts to increase geometrically.

So you can see why, over time, these kinds of tests are going to fail to meet our desiderata because they're too slow to enable a productive workflow and a fast enough feedback cycle.

Don't take it from me! Gary Bernhardt, a legend in both the Ruby and Python testing communities, has a talk simply called "Fast Test, Slow Test", which is a great tour of the problems I'm discussing here.

The Holy Flow State

Thinking sociologically for a moment, we programmers have our own culture and our own "religion" in a way. It has many congregations within it—such as the cult of TDD, to which you are now initiated. There are the followers of Vim and the heretics of Emacs. But one thing we all agree on—one particular spiritual practice, our own transcendental meditation—is the holy flow state. That feeling of pure focus, of concentration, where hours pass like no time at all, where code flows naturally from our fingers, where problems are just tricky enough to be interesting but not so hard that they defeat us…

There is absolutely no hope of achieving flow if you spend your time waiting for a slow test suite to run. Anything longer than a few seconds and you're going to let your attention wander, you context-switch, and the flow state is gone. And the flow state is a fragile dream; once it's gone, it takes a long time to come back.[1]

We're Not Getting the Full Potential Benefits of Testing

TDD experts often say, "It should be called test-driven *design*, not test-driven development". What do they mean by that?

We have definitely seen a bit of the positive influence of TDD on our design. We've talked about how our tests are the first clients of any API we create, and we've talked about the benefits of "programming by wishful thinking" and outside-in.

But there's more to it. These same TDD experts also often say that you should "listen to your tests". Unless you've read the online Appendix: Test Isolation and "Listening to Your Tests", that will still sound like a bit of a mystery.

So, how can we get to a position where our tests are giving us maximum feedback on our design?

1 Some people say it takes at least 15 minutes to get back into the flow state. In my experience, that's overblown, and I sometimes wonder if it's thanks to TDD. I think TDD reduces the cognitive load of programming. By breaking our work down into small increments, by simplifying our thinking—"What's the current failing test? What's the simplest code I can write to make it pass?"—it's often actually quite easy to context-switch back into coding. Maybe it's less true for the times when we're doing design work and thinking about what the abstractions in our code should be though. But also there's absolutely no hope for you if you've started scrolling social media while waiting for your tests to finish. See you in 20 minutes to an hour!

The Ideal of the Test Pyramid

I know I said I didn't want to get bogged down into arguments based on definitions, but let's set out the way people normally think about these three types of tests:

Functional/end-to-end tests
: FTs check that the system works end-to-end, exercising the full stack of the application, including all dependencies and connected external systems. An FT is the ultimate test that it all hangs together, and that things are "really" going to work.

Integration tests
: The purpose of an integration test should be to check that the code you write is integrated correctly with some "external" system or dependency.

(True) unit tests
: Unit tests are the lowest-level tests, and are supposed to test a single "unit" of code or behaviour. The ideal unit test is fully isolated from everything external to the unit under test, such that changes to things outside cannot break the test.

The canonical advice is that you should aim to have the majority of your tests be unit tests, with a smaller number of integration tests, and an even smaller number of functional tests—as in the classic "test pyramid" of Figure 27-1.

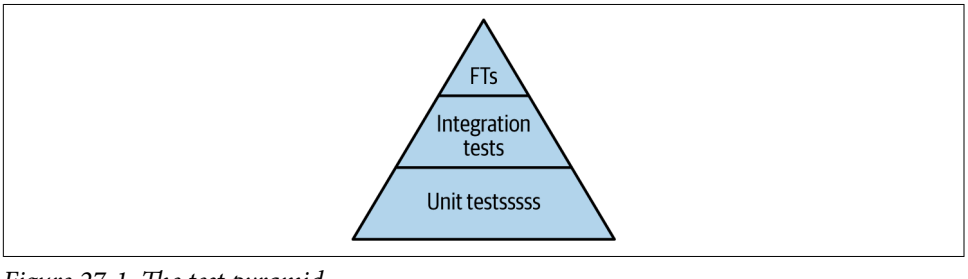


Figure 27-1. The test pyramid

Bottom layer: unit tests (the vast majority)
: These isolated tests are fast and they pinpoint failures precisely. We want these to cover the majority of our functionality, and the entirety of our business logic if possible.

Middle layer: integration tests (a significant portion)
: In an ideal world, these are reserved purely for testing the interactions between our code and external systems—like the database, or even (arguably) Django itself. These are slower, but they give us the confidence that our components work together.

Top layer: a minimal set of functional/end-to-end tests
: These tests are there to give us the ultimate reassurance that everything works end-to-end and top-to-bottom. But because they are the slowest and most brittle, we want as few of them as possible.

On Acceptance Tests

What about "acceptance tests"? You might have heard this term bandied about. Often, people use it to mean the same thing as functional tests or end-to-end tests. But, as taught to me by one of the legends of quality assurance at MADE.com (Hi, Marta!), *any* kind of test can be an acceptance test if it maps onto one of your acceptance criteria.

The point of an acceptance test is to validate a piece of behaviour that's important to the user. In our application, that's how we've been thinking about our FTs.

But, ultimately, using FTs to test every single piece of user-relevant functionality is not sustainable. We need to figure out ways to have our integration tests and unit tests do the work of verifying user-visible behaviour, understood at the right level of abstraction.

Learn more in the video on acceptance test-driven development (ATDD) by Dave Farley.

Avoiding Mock Hell

Well that's all very well, Harry (you might say), but our current test setup is nothing like this! How do we get there from *here*? We've seen how to use mocks to isolate ourselves from external dependencies. Are they the solution then?

As I was at pains to point out the mocking chapters, the use of mocks comes with painful trade-offs:

- They make tests harder to read and write.
- They leave your tests tightly coupled to implementation details.
- As a result, they tend to impede refactoring.
- And, in the extreme, you can sometimes end up with mocks testing mocks, almost entirely disconnected from what the code actually does.

Ed Jung calls this Mock Hell.

This isn't to say that mocks are always bad! But just that, from experience, attempting to use them as your primary tool for decoupling your tests from external dependencies is not a viable solution; it carries costs that often outweigh the benefits.

I'm glossing over the use of mocks in a London-school approach to TDD. See the Online Appendix: Test Isolation and "Listening to Your Tests".

The Actual Solutions Are Architectural

The actual solution to the problem isn't obvious from where we're standing. It lies in rethinking the architecture of our application. In brief, if we can *decouple* the core business logic of our application from its dependencies, then we can write true unit tests for it that do not depend on those, um, dependencies.

Integration tests are most necessary at the *boundaries* of a system—at the points where our code integrates with external systems—like the database, filesystem, network, or a UI. Similarly, it's at the boundaries that the downsides of test isolation and mocks are at their worst, because it's at the boundaries that you're most likely to be annoyed if your tests are tightly coupled to an implementation, or to need more reassurance that things are integrated properly.

Conversely, code at the *core* of our application—code that's purely concerned with our business domain and business rules, code that's entirely under our control—has no intrinsic need for integration tests.

So, the way to get what we want is to minimise the amount of our code that has to deal with boundaries. Then we test our core business logic with unit tests, and test the rest with integration and functional tests.

But how do we do that?

Ports and Adapters/Hexagonal/Onion/Clean Architecture

The classic solutions to this problem from the object-oriented world come under different names, but they're all variations of the same trick: identifying the boundaries, creating an interface to define those boundaries, and then using that interface at test time to swap out fake versions of your real dependencies.

Steve Freeman and Nat Pryce, in their book *Growing Object-Oriented Software, Guided by Tests*, call this approach "Ports and Adapters" (see Figure 27-2).

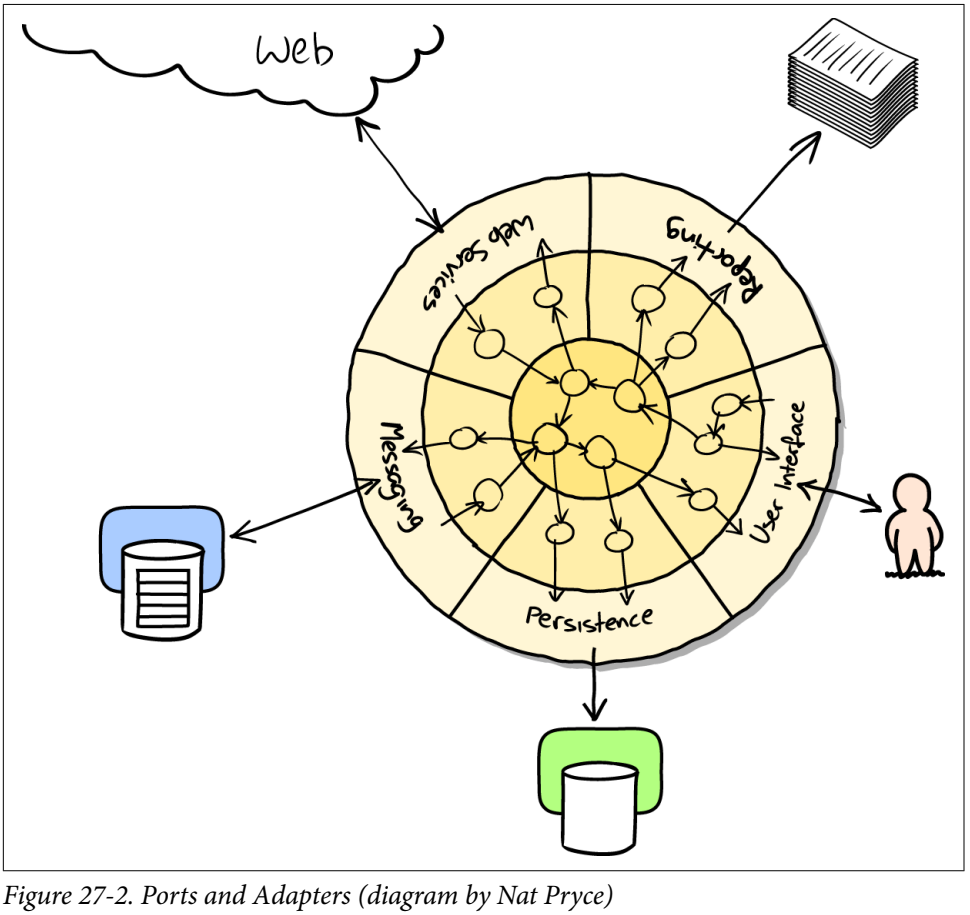


Figure 27-2. Ports and Adapters (diagram by Nat Pryce)

This pattern, or variations of it, are known as "Hexagonal Architecture" (by Alistair Cockburn), "Clean Architecture" (by Robert C. Martin, aka Uncle Bob), or "Onion Architecture" (by Jeffrey Palermo).

Time for a Plug! Read More in "Cosmic Python"

At the end of the process of writing this book (the first time around) I realised that I was going to have to learn about these architectural solutions, and it was at MADE.com that I met Bob Gregory who was to become my coauthor. There, we explored "ports and adapters" and related architectures, which were quite rare at the time in the Python world.

So if you'd like a take on these architectural patterns with a Pythonic twist, check out *Architecture Patterns with Python*, which we subtitled "Cosmic Python", because "cosmos" is the opposite of "chaos", in Greek.

Functional Core, Imperative Shell

Gary Bernhardt pushes this further, recommending an architecture he calls "Functional Core, Imperative Shell", whereby the "shell" of the application (the place where interaction with boundaries happens) follows the imperative programming paradigm, and can be tested by integration tests, functional tests, or even (gasp!) not at all (if it's kept minimal enough).

But the core of the application is actually written following the functional programming paradigm (complete with the "no side effects" corollary), which allows fully isolated, "pure" unit tests—*without any mocks or fakes*.

Check out Gary's presentation titled "Boundaries" for more on this approach.

The Central Conceit: These Architectures Are "Better"

These patterns do not come for free! Introducing the extra indirection and abstraction can add complexity to your code. In fact, the creator of Ruby on Rails, David Heinemeier Hansson (DHH), has a famous blog post where he describes these architectures as test-induced design damage. That post eventually led to quite a thoughtful and nuanced discussion between DHH, Martin Fowler, and Kent Beck.

Like any technique, these patterns can be misused, but I wanted to make the case for their upside: by making our software more testable, we also make it more modular and maintainable. We are forced to clearly separate our concerns, and we make it easier to do things like upgrade our infrastructure when we need to. This is the place where the "improved design" desiderata comes in.

Making our software more testable also often leads to a better design.

Testing in Production

I should also make brief mention of the power of observability and monitoring.

Kent Beck tells a story about his first few weeks at Facebook, when one of the first tests he wrote turned out to be flaky in the build. Someone just deleted it. Shocked and asking why, he was told, "We know production is up. Your test is just producing noise; we don't need it". [2]

Facebook has such confidence in its production monitoring and observability that it can provide them with most of the feedback they need about whether the system is working.

Not everywhere is Facebook! But it's a good indication that automated tests aren't the be-all and end-all.

The Hardest Part: Knowing When to Make the Switch

When is it time to hop out?

For small- to medium-sized applications, as we've seen, the Django test runner and the integration tests it encourages us to write are just fine. The problem is knowing when it's time to make the change to a more decoupled architecture, and to start striving explicitly for the test pyramid.

It's hard to give good advice here, as I've only experienced environments where either someone else made the decision before I joined, or the company is already struggling with a point where it's (at least arguably) too late.

One thing to bear in mind, though, is that the longer you leave it, the harder it is. Another is that because the pain is only going to set in gradually, like the apocryphal boiled frogs, you're unlikely to notice until you're past the "perfect" moment to switch. And on top of that, it's *never* going to be a convenient time to switch. This is one of those things, like tech debt, that is always going to struggle to justify itself in the face of more immediate priorities.

2 There's a transcript of this story.

So, perhaps one strategy would be an Odysseus pact: tie yourself to the mast, and make a commitment—while the tests are still fast—to set a "red line" for when to switch. For example, "If the tests ever take more than 10 seconds to run locally, then it's time to rethink the architecture".

I'm not saying 10 seconds is the right number, by the way. I know plenty of people who are perfectly happy to wait 30 seconds. And I know Gary Bernhardt, for one, would get very nervous at a test suite that takes more than 100 milliseconds.

But I think the idea of drawing that line in the sand, wherever it is, *before* you get there, might be a good way to fight the "boiled frog" problem. Failing all of that, if the best time to make the change was "ages ago", then the second best time is "right now".

Other than that, I can only wish you good luck, and hope that by warning you of the dangers, you'll keep an eye on your test suite and spot the problems before they get too large.

Happy testing!

Wrap-Up

In this book, I've been able to show you how to use TDD, and have talked a bit about why we do it and what makes a good test. But we're inevitably limited by the scope of the project. What that means is that some of the more advanced uses of TDD, particularly the interplay between testing and architecture, have been beyond the scope of this book.

But I hope that this chapter has been a bit a guide to find your way around that topic as your career progresses.

Further Reading

A few places to go for more inspiration:

"Fast Test, Slow Test" and "Boundaries"
: Gary Bernhardt's talks from Pycon 2012 and 2013. His screencasts are also well worth a look.

Integration tests are a scam
: J.B. Rainsberger has a famous rant about the way integration tests will ruin your life.[3] Then check out a couple of follow-up posts, particularly the defence of acceptance tests, and the analysis of how slow tests kill productivity.

Ports and Adapters
: Steve Freeman and Nat Pryce wrote about this in their book. You can also catch a good discussion in Steve's talk. See also Uncle Bob's description of the clean architecture, and Alistair Cockburn coining the term "Hexagonal Architecture".

The test-double testing wiki
: Justin Searls' online resource is a great source of definitions and discussions on testing pros and cons, and arrives at its own conclusions of the right way to do things: testing wiki.

Fowler on unit tests
: Martin Fowler (author of *Refactoring*) offers a balanced and pragmatic tour of what unit tests are, and of the trade-offs around speed.

A take from the world of functional programming
: *Grokking Simplicity* by Eric Normand explores the idea of "Functional Core, Imperative Shell". Don't worry; you don't need a crazy functional programming language like Haskell or Clojure to understand it—it's written in perfectly sensible JavaScript.

3 Rainsberger actually distinguishes "integrated" tests from "integration" tests: an integrated test is any test that's not fully isolated from things outside the unit under test.

Obey the Testing Goat!

Let's get back to the Testing Goat.

"Groan", I hear you say—"Harry, the Testing Goat stopped being funny about 17 chapters ago". Bear with me; I'm going to use it to make a serious point.

Testing Is Hard

I think the reason the phrase "Obey the Testing Goat" first grabbed me when I saw it was that it spoke to the fact that testing is hard—not hard to do in and of itself, but hard to *stick to*, and hard to keep doing.

It always feels easier to cut corners and skip a few tests. And it's doubly hard psychologically because the payoff is so disconnected from the point at which you put in the effort. A test you spend time writing now doesn't reward you immediately; it only helps much later—perhaps months later when it saves you from introducing a bug while refactoring, or catches a regression when you upgrade a dependency. Or, perhaps it pays you back in a way that's hard to measure, by encouraging you to write better-designed code, but you convince yourself you could have written it just as elegantly without tests.

I myself started slipping when I was writing the test framework for this book. Being quite a complex beast, it has tests of its own, but I cut several corners. So, coverage isn't perfect, and I now regret it because it's turned out quite unwieldy and ugly (go on; I've open sourced it now, so you can all point and laugh).

Keep Your CI Builds Green

Another area that takes real hard work is continuous integration. You saw in Chapter 25 that strange and unpredictable bugs sometimes occur in CI. When you're looking at these and thinking "it works fine on my machine", there's a strong temptation to just ignore them…but, if you're not careful, you start to tolerate a failing test suite in CI, and pretty soon your CI build is actually useless, and it feels like too much work to get it going again. Don't fall into that trap. Persist, and you'll find the reason that your test is failing, and you'll find a way to lock it down and make it deterministic, and green, again.

Take Pride in Your Tests, as You Do in Your Code

One of the things that helps is to stop thinking of your tests as being an incidental add-on to the "real" code, and to start thinking of them as being a part of the finished product that you're building—a part that should be just as finely polished and just as aesthetically pleasing, and a part you can be justly proud of delivering…

So, do it because the Testing Goat says so. Do it because you know the payoff will be worth it, even if it's not immediate. Do it out of a sense of duty, or professionalism, or perfectionism, or sheer bloody-mindedness. Do it because it's a good thing to practice. And, eventually, do it because it makes software development more fun.

Remember to Tip the Bar Staff

This book wouldn't have been possible without the backing of my publisher, the wonderful O'Reilly Media. If you're reading the free edition online, I hope you'll consider buying a real copy…if you don't need one for yourself, then maybe as a gift for a friend?

Don't Be a Stranger!

I hope you enjoyed the book. Do get in touch and tell me what you thought!

Harry

- *https://fosstodon.org/@hjwp*
- *obeythetestinggoat@gmail.com*

Bibliography

A few books about TDD and software development that I've mentioned in the book, and which I enthusiastically recommend:

Abelson, Hal, Jerry Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs* (MIT Press, 1996).

Anderson, Ross. *Security Engineering, Third Edition* (Wiley, 2020). *https://www.cl.cam.ac.uk/archive/rja14/book.html*

Beck, Kent. *Test Driven Development: By Example* (Addison-Wesley, 2002).

Farley, Dave. *Modern Software Engineering* (Addison-Wesley, 2021).

Fowler, Martin. *Refactoring* (Addison-Wesley, 2018).

Freeman, Steve and Nat Pryce. *Growing Object-Oriented Software Guided by Tests* (Addison-Wesley, 2009).

APPENDIX A

Cheat Sheet

By popular demand, this "cheat sheet" is loosely based on the recap/summary boxes from the end of each chapter. The idea is to provide a few reminders, and links to the chapters where you can find out more to jog your memory. I hope you find it useful!

Initial Project Setup

- Start with a *user story* and map it to a first *functional test*.
- Pick a test framework—`unittest` is fine, and options like `py.test`, `nose`, or `Green` can also offer some advantages.
- Run the functional test and see your first *expected failure*.
- Pick a web framework such as Django, and find out how to run *unit tests* against it.
- Create your first *unit test* to address the current FT failure, and see it fail.
- Do your *first commit* to a VCS like *Git*.

Relevant chapters: Chapter 1, Chapter 2, Chapter 3.

The Basic TDD Workflow: Red/Green/Refactor

- Red, Green, Refactor
- Double-loop TDD (Figure A-1)
- Triangulation
- The scratchpad
- "3 Strikes and Refactor"
- "Working State to Working State"
- "YAGNI"

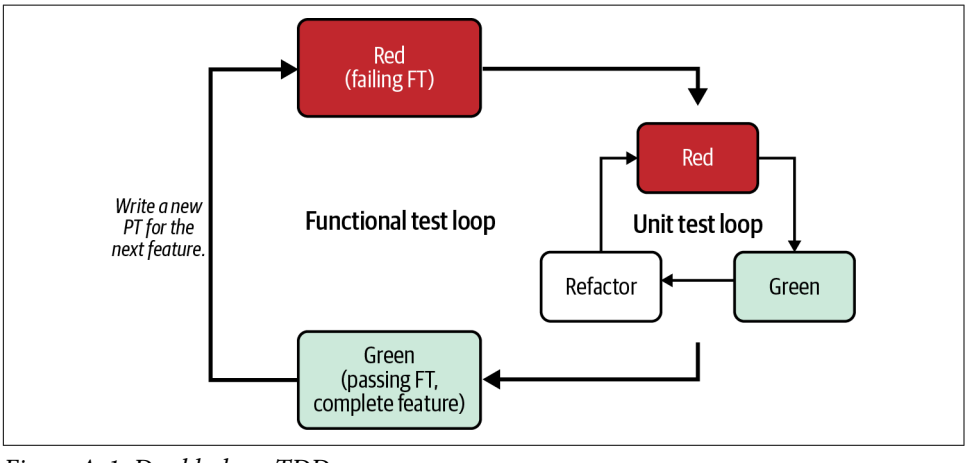


Figure A-1. Double-loop TDD

Relevant chapters: Chapter 4, Chapter 5, Chapter 7.

Moving Beyond Dev-Only Testing

- Start system testing early. Ensure your components work together: web server, static content, database.
- Build a production environment early, and automate deployment to it.
  - — PaaS versus VPS
  - — Docker
  - — Ansible versus Terraform
- Think through deployment pain points: the database, static files, dependencies, how to customise settings, and so on.
- Build a CI server as soon as possible, so that you don't have to rely on self-discipline to see the tests run.

Relevant chapters: Part II, Chapter 25.

General Testing Best Practices

- Each test should test one thing.
- Test behaviour rather than implementation.
- "Don't test constants".
- Try to think beyond the charmed path through the code, and think through edge cases and error cases.
- Balance the "test desiderata".

Relevant chapters: Chapter 4, Chapter 14, Chapter 15, Chapter 27.

Selenium/Functional Testing Best Practices

- Use explicit rather than implicit waits, and the interaction/wait pattern.
- Avoid duplication of test code--helper methods in a base class and the page pattern are possible solutions.
- Avoid double-testing functionality. If you have a test that covers a time-consuming process (e.g., login), consider ways of skipping it in other tests (but be aware of unexpected interactions between seemingly unrelated bits of functionality).
- Look into BDD tools as another way of structuring your FTs.

Relevant chapters: Chapter 23, Chapter 25, Chapter 26.

Outside-In

Default to working outside-in. Use double-loop TDD to drive your development, start at the UI/outside layers, and work your way down to the infrastructure layers. This helps ensure that you write only the code you need, and flushes out integration issues early.

Relevant chapter: Chapter 24.

The Test Pyramid

Be aware that integration tests will get slower and slower over time. Find ways to shift the bulk of your testing to unit tests as your project grows in size and complexity.

Relevant chapter: Chapter 27.

APPENDIX B

What to Do Next

Here I offer a few suggestions for things to investigate next, to develop your testing skills, and to apply them to some of the cool new technologies in web development (at the time of writing!).

I might write an article about some of these in the future. But why not try to beat me to it, and write your own blog post chronicling your attempt at any one of these?

I'm very happy to answer questions and provide tips and guidance on all these topics, so if you find yourself attempting one and getting stuck, please don't hesitate to get in touch at *obeythetestinggoat@gmail.com*!

Switch to Postgres

SQLite is a wonderful little database, but it won't deal well once you have more than one web worker process fielding your site's requests. Postgres is everyone's favourite database these days, so find out how to install and configure it.

You'll need to figure out a place to store the usernames and passwords for your local, staging, and production Postgres servers. Take a look at Chapter 12 for inspiration.

Experiment with keeping your unit tests running with SQLite, and compare how much faster they are than running against Postgres. Set it up so that your local machine uses SQLite for testing, but your CI server uses Postgres.

Does any of your functionality actually depend on Postgres-specific features? What should you do then?

Run Your Tests Against Different Browsers

Selenium supports all sorts of different browsers, including Chrome, Safari, and Internet Exploder. Try them all out and see if your FT suite behaves any differently.

In my experience, switching browsers tends to expose all sorts of race conditions in Selenium tests, and you will probably need to use the interaction/wait pattern a lot more.

The Django Admin Site

Imagine a story where a user emails you wanting to "claim" an anonymous list. Let's say we implement a manual solution to this, involving the site administrator manually changing the record using the Django admin site.

Find out how to switch on the admin site, and have a play with it. Write an FT that shows a normal, non–logged-in user creating a list, then have an admin user log in, go to the admin site, and assign the list to the user. The user can then see it in their "My Lists" page.

Write Some Security Tests

Expand on the login, my lists, and sharing tests—what do you need to write to assure yourself that users can only do what they're authorized to?

Test for Graceful Degradation

What would happen if our email server goes down? Can we at least show an apologetic error message to our users?

Caching and Performance Testing

Find out how to install and configure `memcached`. Find out how to use Apache's `ab` to run a performance test. How does it perform with and without caching? Can you write an automated test that will fail if caching is not enabled? What about the dreaded problem of cache invalidation? Can tests help you to make sure your cache invalidation logic is solid?

JavaScript Frameworks

Check out React, Vue.js, or perhaps my old favourite, Elm.

Async and Websockets

Supposing two users are working on the same list at the same time. Wouldn't it be nice to see real-time updates, so if the other person adds an item to the list, you see it immediately? A persistent connection between client and server using websockets is the way to get this to work.

Check out Django's async features and see if you can use them to implement dynamic notifications.

To test it, you'll need two browser instances (like we used for the list sharing tests), and check that notifications of the actions from one appear in the other, without needing to refresh the page…

Switch to Using pytest

`pytest` lets you write unit tests with less boilerplate. Try converting some of your unit tests to using *py.test*. You may need to use a plugin to get it to play nicely with Django.

Check Out coverage.py

Ned Batchelder's `coverage.py` will tell you what your *test coverage* is—what percentage of your code is covered by tests. Now, in theory, because we've been using rigorous TDD, we should always have 100% coverage. But it's nice to know for sure, and it's also a very useful tool for working on projects that didn't have tests from the beginning.

Client-Side Encryption

Here's a fun one: what if our users are paranoid about the NSA, and decide they no longer want to trust their lists to The Cloud? Can you build a JavaScript encryption system, where the user can enter a password to encypher their list item text before it gets sent to the server?

One way of testing it might be to have an "administrator" user that goes to the Django admin view to inspect users' lists, and checks that they are stored encrypted in the database.

Your Suggestion Here

What do you think I should put here? Suggestions, please!

APPENDIX C

Source Code Examples

All of the code examples I've used in the book are available in my repo on GitHub. So, if you ever want to compare your code against mine, you can take a look at it there.

Each chapter has its own branch named after it, like so: *https://github.com/hjwp/book-example/tree/chapter_01*.

Be aware that each branch contains all of the commits for that chapter, so its state represents the code at the *end* of the chapter.

Full List of Links for Each Chapter

Chapter	GitHub branch name & hyperlink
Chapter 1	chapter_01
Chapter 2	chapter_02_unittest
Chapter 3	chapter_03_unit_test_first_view
Chapter 4	chapter_04_philosophy_and_refactoring
Chapter 5	chapter_05_post_and_database
Chapter 6	chapter_06_explicit_waits_1
Chapter 7	chapter_07_working_incrementally
Chapter 8	chapter_08_prettification
Chapter 9	chapter_09_docker
Chapter 10	chapter_10_production_readiness
Chapter 11	chapter_11_server_prep
Chapter 13	chapter_13_organising_test_files
Chapter 14	chapter_14_database_layer_validation
Chapter 15	chapter_15_simple_form

Chapter	GitHub branch name & hyperlink
Chapter 16	chapter_16_advanced_forms
Chapter 17	chapter_17_javascript
Chapter 18	chapter_18_second_deploy
Chapter 19	chapter_19_spiking_custom_auth
Chapter 20	chapter_20_mocking_1
Chapter 21	chapter_21_mocking_2
Chapter 22	chapter_22_fixtures_and_wait_decorator
Chapter 23	chapter_23_debugging_prod
Chapter 24	chapter_24_outside_in
Chapter 25	chapter_25_CI
Chapter 26	chapter_26_page_pattern
Online Appendix: Test Isolation, and Listening to Your Tests	appendix_purist_unit_tests
Online Appendix: BDD	appendix_bdd
Online Appendix: Building a REST API	appendix_rest_api

Using Git to Check Your Progress

If you feel like developing your Git-Fu a little further, you can add my repo as a *remote*:

```
git remote add harry https://github.com/hjwp/book-example.git
git fetch harry
```

And then, to check your difference from the *end* of Chapter 4:

```
git diff harry/chapter_04_philosophy_and_refactoring
```

Git can handle multiple remotes, so you can still do this even if you're already pushing your code up to GitHub or Bitbucket.

Be aware that the precise order of, say, methods in a class may differ between your version and mine. It may make diffs hard to read.

Downloading a ZIP File for a Chapter

If, for whatever reason, you want to "start from scratch" for a chapter, or skip ahead,[1] and/or you're just not comfortable with Git, you can download a version of my code as a ZIP file, from URLs following this pattern:

https://github.com/hjwp/book-example/archive/chapter_01.zip

1 I don't recommend skipping ahead. I haven't designed the chapters to stand on their own; each relies on the previous ones, so it may be more confusing than anything else…

https://github.com/hjwp/book-example/archive/chapter_04_philosophy_and_refactoring.zip

Don't Let it Become a Crutch!

Try not to sneak a peek at the answers unless you're really, really stuck. Like I said at the beginning of Chapter 3, there's a lot of value in debugging errors all by yourself, and in real life, there's no "harrys repo" to check against and find all the answers.

Happy coding!

Index

Symbols

A

D

E

F

G

K

L

M

T

U

V

W

Y

About the Author

After an idyllic childhood spent playing with BASIC on French 8-bit computers like the Thomson T07 whose keys go "boop" when you press them, **Harry Percival** spent a few years being deeply unhappy with economics and management consultancy. Soon, he rediscovered his true geek nature, and was lucky enough to fall in with a bunch of XP fanatics, working on the pioneering but sadly defunct Resolver One spreadsheet. He now works at PythonAnywhere LLP, and spreads the gospel of test-driven development worldwide at talks, workshops, and conferences, with all the passion and enthusiasm of a recent convert.

Colophon

The animal on the cover of *Test-Driven Development with Python* is a cashmere goat. Though all goats can produce a cashmere undercoat, only those goats selectively bred to produce cashmere in commercially viable amounts are typically considered "cashmere goats". Cashmere goats thus belong to the domestic goat species *Capra hircus*.

The exceptionally fine, soft hair of the undercoat of a cashmere goat grows alongside an outercoat of coarser hair as part of the goat's double fleece. The cashmere undercoat appears in winter to supplement the protection offered by the outercoat, called "guard hair". The crimped quality of cashmere hair in the undercoat accounts for its lightweight yet effective insulation properties.

The name "cashmere" is derived from the Kashmir Valley region on the Indian subcontinent, where the textile has been manufactured for thousands of years. A diminishing population of cashmere goats in modern Kashmir has led to the cessation of exports of cashmere fiber from the area. Most cashmere wool now originates in Afghanistan, Iran, Outer Mongolia, India, and—predominantly—China.

Cashmere goats grow hair of varying colors and color combinations. Both males and females have horns, which serve to keep the animals cool in summer and provide the goats' owners with effective handles during farming activities.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover image is from Wood's Animate Creation. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; the code font is Dalton Maag's Ubuntu Mono; and the Scratchpad font is ORAHand-Medium.