



The Django Admin Handbook

The Ultimate Guide to Building Your Back
Office in Hours, Not Days

Christophe Surbier

Apress®

The Django Admin Handbook

**The Ultimate Guide to Building Your
Back Office in Hours, Not Days**

Christophe Surbier

Apress®

The Django Admin Handbook: The Ultimate Guide to Building Your Back Office in Hours, Not Days

Christophe Surbier
Barcelona, Spain

ISBN-13 (pbk): 979-8-8688-2899-7

ISBN-13 (electronic): 979-8-8688-2900-0

<https://doi.org/10.1007/979-8-8688-2900-0>

Copyright © 2026 by Christophe Surbier

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr

Acquisitions Editor: James Robinson-Prior

Editorial Assistant: Gryffin Winkler

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit <https://www.apress.com/gp/services/source-code>.

If disposing of this product, please recycle the paper

Table of Contents

About the Author xi

About the Technical Reviewer xiii

Acknowledgments xv

Part I: Foundations 1

Chapter 1: Introduction and Setup 3

 What Is Django Admin? 3

 Who Is This Book For?..... 4

 What Will We Learn? 4

 Requirements..... 5

 Installation 5

 Project Structure..... 8

Chapter 2: Core Concepts 9

 Models and Registration 10

 Designing Models 10

 Registering Our Models in the Django Admin Interface..... 17

 Creating Fake Data 20

Chapter 3: Basic Configuration..... 25

 Model 25

 Admin 26

 Customizing the Admin Interface 27

 Internationalization 29

 Summary..... 33

Part II: Essential Features..... 35

Chapter 4: Display and Filtering 37

 List Display Options 37

 Date 38

 Customizing Field Names 39

 Displaying a Field from a Related Model 43

 list_per_page 46

 Ordering Instruction..... 47

 list_display_links..... 49

 list_editable 51

 Filter on the Change List..... 52

 Related Field Filters..... 54

 Custom Filters 55

 Search the Change List..... 60

 Search Fields Help Text 62

 Change Form..... 62

 fields Attribute 62

 readonly_fields 63

 exclude 64

 fieldsets..... 66

Chapter 5: Actions and Operations 73

 Built-In Actions..... 73

 Custom Actions 74

 Action Responses and Messages 76

Chapter 6: Forms and Validation 79

 Form Customization 79

 Validation Rules 81

Chapter 7: Inline Editing	83
Many to Many Relationships.....	86
Chapter 8: Performance Optimization	93
Query Optimization	93
select_related.....	93
prefetch_related	102
Simple Caching Strategies.....	103
Pagination Handling	106
Basic Database Indexes	107
Summary.....	108
Part III: Advanced Customization	115
Chapter 9: Extend Features	117
list_display.....	117
list_display: Display Custom Link.....	117
list_display: Custom Admin URLs and Actions.....	121
list_display: Chain Filter	125
list_display: Display a Database View	140
list_display: Proxying a Model.....	141
search_fields: custom get_search_results().....	142
change_view: Display a change_form with Prefilled Values	146
change_view: Forms Widget.....	149
change_view: Display a change_form with Foreign Key Restriction	151
change_view: Initialize an Inline Formset and Do Some Custom Validation.....	152
change_view: raw_id_fields.....	156
change_view: Inline Pagination	158
change_view: Some Restrictions.....	160
Add Actions with Intermediate Page	161

TABLE OF CONTENTS

Chapter 10: Template System Inheritance..... 169

 Change Login Form 173

 Change Default Ordering of Our Models in the Admin Site 180

 Change Default Admin Index Page..... 183

 Change List View..... 189

 Change View: Make an Ajax Request 195

Chapter 11: Custom Views and URLs..... 209

 Ajax Integration in Admin 209

 Faking Models..... 211

 Advanced URL Management in Django Admin 226

 Key Points About Custom URLs 227

Chapter 12: REST API Integration with Django Admin..... 229

 API Endpoints 229

 Serialization 232

Chapter 13: Background Tasks..... 237

 Celery Integration..... 237

 Task Management..... 246

Chapter 14: Advanced Query Optimization..... 251

 Raw SQL Optimization When Needed 251

 Query Debugging and Profiling Tools 253

 Summary..... 256

Part IV: UI/UX Enhancement 257

Chapter 15: Template System Deep Dive 259

 Block Override Patterns 259

 Template Inheritance Hierarchy..... 259

 In Template base.html 259

 In template index.html..... 261

 How Block Tags Work in Template Inheritance 263

Template Resolution Order	264
Best Practices for Working with Block Tags	265
Custom Tags.....	265
Why Use Custom Tags in Django Admin?	265
Creating Custom Template Tags	266
Using Custom Tags in Admin Templates	268
Best Practices for Custom Tags	271
Custom Filters	272
Creating Custom Filters	272
Using Custom Filters in Templates	273
Existing Packages to Rebrand Django Admin	276
Django Jet	276
Django Grapelli	277
Django Jazzmin	278
Django Daisy.....	279
Django Unfold	280
Chapter 16: Modern CSS Frameworks.....	285
Bootstrap Integration	285
Bootstrap cdn Version.....	285
Bootstrap Locally.....	287
Tailwind CSS	289
Material Design.....	293
Custom Themes	294
Chapter 17: JavaScript Integration	307
HTMX and Ajax Operations.....	308
Create a View with Recent Orders for Our HTMX Component	315
Add URLs.....	315
Create a Template for Recent Orders Display	316
Update the HTMX Component with Our New Endpoint and with an Auto-Refresh Instruction	319

TABLE OF CONTENTS

Chapter 18: Responsive Design 323

 Mobile-First Approach..... 323

 Adaptive Layouts..... 323

 Media Queries 323

Chapter 19: Custom Dashboards 327

 TODO Models..... 329

 Create Views for TODO Widget 331

 Create TODO Widget Template..... 333

 Integrate TODO Widget into Admin Index 338

Chapter 20: Charts and Analytics 341

 Add Chart.js Integration to Order Change List Template 342

 Create View for Order Chart Data..... 346

 Alternative: Build the Same Charts with json_script (No Extra API Call)..... 347

 1. Build the Chart Payload As a Python dict..... 347

 2. Add the Payload to the changelist Context..... 348

 3. Embed the JSON in the Template with json_script, then Parse It 349

 Report Generation 349

 CSV Export..... 349

 PDF Reports..... 354

 Avoiding CDNs: Serving Frontend Assets with Django staticfiles 364

 Put the Asset in Your Project 364

 Reference It in Templates with {% static %}..... 364

 Settings and Deployment Notes 365

 Summary..... 365

Part V: Security and Access Control 367

Chapter 21: Protect Django Admin Access 369

 Customizing the Admin URL..... 369

 IP Blocking/Filtering 371

 Brute-Force Protection (Login Lockout) 373

 HTTPS and SSL/TLS Encryption 375

Two-Factor Authentication	375
Protect SECRET_KEY	379
Why SECRET_KEY Is Critical	379
Common SECRET_KEY Exposure Risks	380
Best Practices for SECRET_KEY Management	381
Emergency Response	382
Example Implementation.....	383
Chapter 22: Permission Systems.....	385
Permission Models.....	385
Creating a New User.....	387
Group Permissions	392
Chapter 23: Role-Based Access Control	399
Row-Level Access (get_queryset).....	402
Custom Permissions	403
Chapter 24: Audit Logging	407
Audit Log	408
Chapter 25: Security Best Practices.....	413
CSRF Protection	413
XSS Prevention.....	413
SQL Injection	414
Security Headers.....	414
Session Security	414
Password Policy.....	415
Production Security Checklist.....	415
Summary.....	415
Conclusion	417
Index.....	421

About the Author



Christophe Surbier is a freelancer, expert in mobile application development with more than 15 years of expertise. Since 2009, he has architected and delivered over 200 mobile applications, starting with native iOS development before strategically transitioning to hybrid solutions with Ionic framework and Capacitor in 2017. This extensive portfolio demonstrates his deep understanding of mobile development ecosystems and his ability to adapt to evolving technologies for more efficient cross-platform delivery.

Throughout his career, Christophe has specialized in creating robust backend systems using Python Django, particularly focusing on REST API architectures and Django Admin implementations for efficiently managing mobile applications and business operations.

In 2025, he founded SellTix, a secure ticketing platform for online events built on Angular and Django for backend, featuring tamper-proof digital tickets and integrated fraud protection.

About the Technical Reviewer



Dmytro Knivets is a software engineer with over a decade of experience building web applications with Django, with a focus on performance and application security. He has contributed to Django REST Framework and has worked in industries such as healthcare and fintech.

Acknowledgments

I would like to thank Frédéric Lempereur for introducing me to Python and Django back in 2004, which was a true revelation, and Dorothée Schved for all the years we spent working together. And finally, a heartfelt thank you to everyone who contributes, near or far, to the wonderful Django framework—with a special shout-out to the Django Admin contributors.

PART I

Foundations

This part introduces Django Admin, covering installation, key concepts, and initial configuration.

CHAPTER 1

Introduction and Setup

What Is Django Admin?

Django Admin provides a powerful and flexible way to quickly build a professional backoffice interface for managing your application's data and business operations. It allows developers to quickly create sophisticated and solid administrative interfaces with minimal code while offering extensive customization options.

At its core, Django Admin works by reading model metadata and configuration files to automatically generate a model-centric interface. This interface allows trusted users to perform CRUD operations (Create, Read, Update, Delete) on your site's content with minimal setup required. The Admin site interprets your models' fields, relationships, and metadata to create intuitive forms and list views, making it an incredibly efficient tool for rapid backoffice development.

While CRUD operations form the foundation, Django Admin extends far beyond basic data management. Through its rich customization API, developers can implement business logic, add custom actions for batch operations, create sophisticated filters, integrate with external services, and build tailored workflows that match specific business requirements. This flexibility transforms Django Admin from a simple data management tool into a comprehensive business operations platform.

However, when projects require highly specialized workflows or extensive customizations, developing a custom backoffice solution may become more practical. This book will guide you through Django Admin's full capabilities and limitations, enabling you to make an informed decision about whether it's the right choice for your specific use case. By understanding both its strengths and boundaries, you'll be equipped to determine if Django Admin can effectively serve your project's needs or if a custom solution would be more appropriate.

Who Is This Book For?

- Django developers who want to rapidly build professional administrative interfaces and need to understand the full potential of the Django Admin framework
- Backend engineers working on business applications who need to provide quickly powerful backoffice capabilities for their organization's staff
- Full-stack developers looking to implement data management interfaces while maintaining Django's rapid development philosophy
- Technical leads and architects evaluating Django Admin's capabilities for enterprise projects and need to make informed decisions about its adoption
- Experienced Django developers seeking to extend their knowledge of admin customization, performance optimization, and security best practices

What Will We Learn?

- Master Django Admin customization techniques to build professional backoffice interfaces in hours instead of days
- Implement advanced features including custom views, REST API integration, and sophisticated query optimization
- Transform the default interface into a modern, responsive application using contemporary CSS frameworks such as Bootstrap or Tailwind
- Build powerful batch processing systems with custom admin actions and Celery integration for handling resource-intensive operations and long-running tasks
- Design and implement enterprise-grade security systems with role-based access control

All source code of this book can be found on Github and downloaded from <https://github.com/Apress/The-Django-Admin-Playbook>.

Requirements

This book has been written using the following software version:

Python 3.10 and **Django 5.1**

Installation

This book assumes you have Python and virtualenv already installed on your computer.

First we need to install our virtual environment:

```
virtualenv -p python3.10 venv
```

and activate it:

```
source venv/bin/activate
```

Now we install Django:

```
pip install django==5.1
```

And create our Django project:

```
django-admin startproject djangoadmin
```

Now we will create our application and call it **backoffice**. To do so, just go to the created new folder **djangoadmin** and run the following command:

```
python manage.py startapp backoffice
```

Django comes with a built-in SQLite database by default, which is perfect for development and learning purposes. SQLite is a lightweight, file-based database that requires no separate server process. Before we can use our Django project, we need to create the initial database structure by running migrations. Migrations are Django's way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. The initial migrations will create essential tables for Django's authentication system and session management. To create these tables, run the following command:

```
python manage.py migrate
```

🔗 Each time you will make a change to our models such as adding a new model or modifying an existing one, we will need to run the migration command again:

```
python manage.py makemigrations
python manage.py migrate
```

This ensures that our database schema remains up-to-date with our code changes. The migration process will create a **migrations** folder in our backoffice project directory. This folder will contain the migration files that Django uses to keep track of changes. If you're curious about the details of the migrations, you can view them by running

```
python manage.py showmigrations
```

This command will show us the status of our migrations. At any moment, we can see the migrations that have been applied to the database (marked with an [X]).

```
admin
```

```
[X] 0001_initial
[X] 0002_logentry_remove_auto_add
[X] 0003_logentry_add_action_flag_choicesauth
[X] 0001_initial
[X] 0002_alter_permission_name_max_length
[X] 0003_alter_user_email_max_length
[X] 0004_alter_user_username_opts
[X] 0005_alter_user_last_login_null
[X] 0006_require_contenttypes_0002
[X] 0007_alter_validators_add_error_messages
[X] 0008_alter_user_username_max_length
[X] 0009_alter_user_last_name_max_length
[X] 0010_alter_group_name_max_length
[X] 0011_update_proxy_permissions
[X] 0012_alter_user_first_name_max_lengthcontenttypes
```

```
[X] 0001_initial  
[X] 0002_remove_content_type_name  
[X] 0001_initial
```

🔗 Migrations which have not been applied to the database are marked with a [].

Then we need to create a superuser that will be used to access our Django Admin interface:

```
python manage.py createsuperuser
```

We will have to choose a username, an email, and a robust password. Once done, we can access the Django Admin interface by running the following command which will launch the Django development server:

```
python manage.py runserver
```

and open a browser at the URL <http://127.0.0.1:8000/admin> to see the login form.

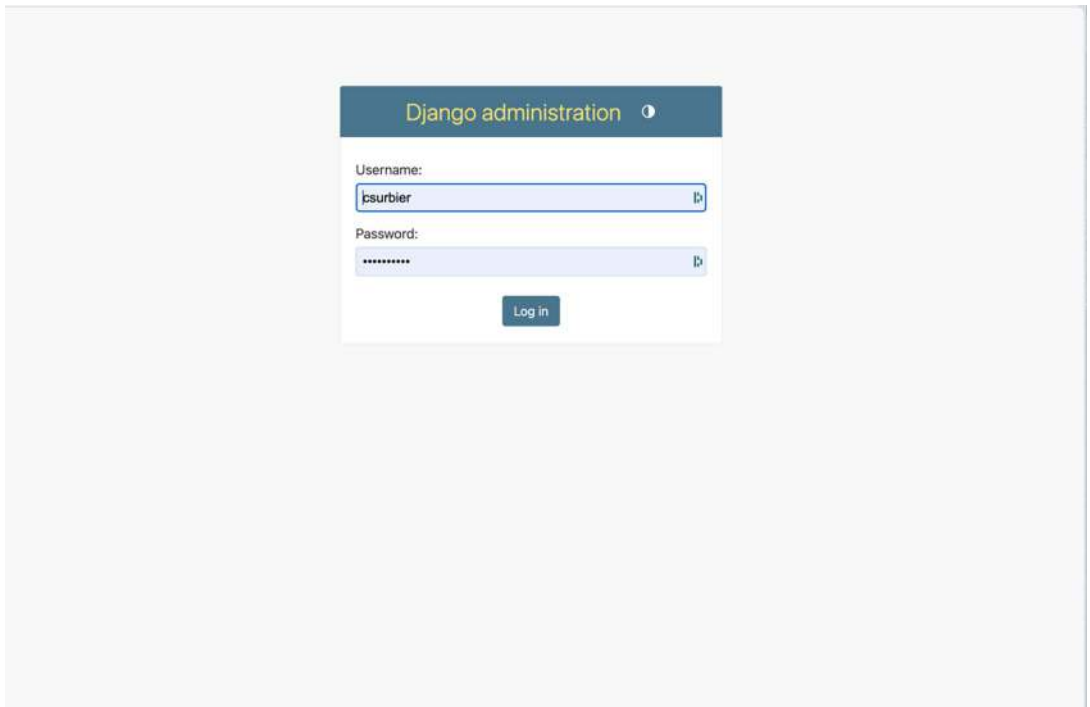


Figure 1-1. Django Admin login form

Project Structure

The `djangoadmin` directory contains our project with settings and configuration files.

The `backoffice` application will contain all the required files to set up our admin interface such as models, views, templates, and much more. We'll be working primarily with

- `models.py`: File in which we are defining our database schema
- `admin.py`: File in which we'll register and customize our database models' admin interfaces
- `templates/admin/`: For overriding default admin templates
- `static/`: For storing static files like CSS and JavaScript

CHAPTER 2

Core Concepts

Let's explore the fundamental architecture and key components that make Django Admin such a powerful tool for building backoffice interfaces. We will review each of these concepts throughout this book:

- **ModelAdmin Classes:** These classes define how models are displayed and managed in the admin interface. They encapsulate the logic for displaying, editing, and managing model instances.
- **AdminSite Class:** This class is the entry point for the admin interface. It manages the registration of models and provides the interface for interacting with the admin site.
- **URL Routing:** Django Admin uses URL routing to map URLs to specific views and actions. This allows users to navigate and interact with the admin interface.
- **Template Inheritance:** Django Admin leverages template inheritance to provide a consistent and customizable user interface. This allows us to override and extend the default templates to match our project's design and functionality.
- **Asset Management:** Django Admin handles the management of static assets like CSS, JavaScript, and images. This ensures that the admin interface is visually appealing and responsive.
- **Customization:** Django Admin allows us to customize the admin interface to fit our project's needs. This includes adding custom actions, filters, and views.
- **Integration:** Django Admin seamlessly integrates with other Django components, such as forms, views, and models, making it a powerful tool for building complex backoffice interfaces.

- **Extensibility:** Django Admin is highly extensible. We can create custom actions, filters, and views to tailor the admin interface to our specific needs.
- **Internationalization:** Django Admin supports internationalization, allowing us to localize the admin interface for users with different languages and locales.
- **Security:** Django Admin provides robust security features to protect the admin interface from unauthorized access and ensure data integrity.

Models and Registration

In the `models.py` file of our `backoffice` folder, we define our database schema.

Within this book, we will study a real-world example and not just a simple TODO tasks management system. So let's study **an Uber-Eats-like system** that will allow us to manage users, restaurants, dishes, and orders, and later we will also learn how to add geolocation features in our Admin.

Designing Models

Let's edit the `models.py` file and define our models:

```
from django.db import models
from django.contrib.auth.models import User
from django.utils.translation import gettext_lazy as _ # Use this to get
translated strings

class Restaurant(models.Model):
    name = models.CharField(_('name'), max_length=200)
    image = models.ImageField(upload_to='restaurants/', blank=True,
    null=True)
    address = models.TextField(_('address'))
    zip_code = models.CharField(_('zip code'), max_length=20)
    city = models.CharField(_('city'), max_length=100)
    country = models.CharField(_('country'), max_length=100)
```

```

phone = models.CharField(_('phone'), max_length=20)
owner = models.ForeignKey(User, on_delete=models.PROTECT,
related_name='restaurants')
is_active = models.BooleanField(_('active'), default=True)
created_at = models.DateTimeField(auto_now_add=True)

```

```

class Meta:

```

```

    verbose_name = _('restaurant')
    verbose_name_plural = _('restaurants')
    ordering = ['-created_at']

```

```

def __str__(self):

```

```

    return self.name

```

```

class Category(models.Model):

```

```

    name = models.CharField(_('name'), max_length=100)
    picto = models.ImageField(upload_to='restaurants/', blank=True,
null=True)
    restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE,
related_name='categories')
    is_active = models.BooleanField(_('active'), default=True)
    created_at = models.DateTimeField(auto_now_add=True)

```

```

class Meta:

```

```

    verbose_name = _('category')
    verbose_name_plural = _('categories')
    ordering = ['name']

```

```

def __str__(self):

```

```

    return f"{self.restaurant.name} - {self.name}"

```

```

class Dish(models.Model):

```

```

    name = models.CharField(_('name'), max_length=200)
    description = models.TextField(_('description'))
    price = models.DecimalField(_('price'), max_digits=6, decimal_places=2)
    category = models.ForeignKey(Category, on_delete=models.CASCADE,
related_name='dishes')
    is_available = models.BooleanField(_('available'), default=True)
    image = models.ImageField(_('image'), upload_to='dishes/', blank=True)

```

```

class Meta:
    verbose_name = _('dish')
    verbose_name_plural = _('dishes')
    ordering = ['name']

def __str__(self):
    return f"{self.category.restaurant.name} - {self.name}"

```

```

class Order(models.Model):
    STATUS_CHOICES = [
        ('pending', _('Pending')),
        ('confirmed', _('Confirmed')),
        ('preparing', _('Preparing')),
        ('delivering', _('Delivering')),
        ('delivered', _('Delivered')),
        ('cancelled', _('Cancelled')),
    ]

    customer = models.ForeignKey(User, on_delete=models.PROTECT, related_name='orders')
    restaurant = models.ForeignKey(Restaurant, on_delete=models.PROTECT, related_name='orders')
    status = models.CharField(_('status'), max_length=20, choices=STATUS_CHOICES, default='pending')
    delivery_address = models.TextField(_('delivery address'))
    created_at = models.DateTimeField(auto_now_add=True)
    total_amount = models.DecimalField(_('total amount'), max_digits=8, decimal_places=2)

```

```

class Meta:
    verbose_name = _('order')
    verbose_name_plural = _('orders')
    ordering = ['-created_at']

```



```

def __str__(self):
    return f"Order #{self.id} - {self.restaurant.name}"

class OrderItem(models.Model):
    order = models.ForeignKey(Order, on_delete=models.CASCADE, related_
name='items')
    dish = models.ForeignKey(Dish, on_delete=models.PROTECT, related_
name='order_items')
    quantity = models.PositiveIntegerField(_('quantity'))
    unit_price = models.DecimalField(_('unit price'), max_digits=6,
decimal_places=2)

    class Meta:
        verbose_name = _('order item')
        verbose_name_plural = _('order items')

    def __str__(self):
        return f"{self.order.id} - {self.dish.name} x {self.quantity}"

```

Each model defines a `__str__(self)` method, which returns a human-readable string representation of an object. Django uses this in many places, especially in the admin: object dropdowns (foreign keys), the “breadcrumbs” at the top of change pages, and various lists where Django needs to display an instance as text.

Let’s review the models and their purpose (Table 2-1).

Table 2-1. *Backoffice domain models and their purpose*

Model	Description
Restaurant	Food establishment Tracks basic business information (name, address, phone) Links to an owner (User) with PROTECT to maintain data integrity Includes activation status and creation timestamp for business operations
Category	Organizes dishes into logical groups (e.g., appetizers, main courses, desserts) Belongs to a specific restaurant using CASCADE deletion Maintains active status for seasonal or temporary categories Uses natural ordering by name for consistent display
Dish	Represents individual menu items Contains essential food details: name, description, price Links to a category with CASCADE for automatic cleanup Includes availability flag for managing menu status Supports image uploads for visual presentation
Order	Tracks customer purchases Uses status choices for order lifecycle management Links both customer and restaurant with PROTECT Maintains delivery information and total amount Orders by creation date for chronological display
OrderItem	Represents individual items within an order Links to both order (CASCADE) and dish (PROTECT) Stores quantity and unit price at time of purchase Enables order history and financial tracking

🔗 The `gettext_lazy` as `_` import and usage of `_()` around strings enables Django’s internationalization system. This allows our admin interface to be translated into different languages. We’ll explore translation features at the end of this chapter.

To enable our application, we need to register our **backoffice** application inside the list of `INSTALLED_APPS` which is defined in the `settings.py` file of our `djangoadmin` project.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'backoffice',
]
```

This step is essential because it tells Django to

- Include and recognize our `backoffice` app as part of the project
- Look for `models.py`, `admin.py`, and other Django-specific files in the `backoffice` directory
- Include the app's database models when running migrations
- Load templates and static files from the app's directories
- Make the app's admin customizations available to Django Admin

Without registering it in `INSTALLED_APPS`, Django would not be able to find and use any of our `backoffice` app's components.

Before creating our models effectively in our database, we need to install some libraries. We will create a `requirements.txt` file in our project directory. This file will contain all the libraries we need to install to run our project, with their versions. To get the list of the libraries we already installed, we can run the following command:

```
pip freeze > requirements.txt
```

which will output something like this in our terminal:

```
asgiref==3.8.1
Django==5.1
sqlparse==0.5.3
```

Please note that the output will be different depending on the python version you are using.

Now to manage our **ImageField**, we need to install Pillow library:

```
pip install pillow
```

And if we run again the `pip freeze` command, we will see the following output:

```
asgiref==3.8.1
Django==5.1
sqlparse==0.5.3
pillow==11.1.0
```

🐘 Specifying the library versions helps track which version was used during development. This is crucial because different versions of a library may have different features or incompatibilities. Having exact versions documented ensures reliable production maintenance and consistent behavior across all environments.

Now we can create our models by preparing the migrations:

```
python manage.py makemigrations
```

We will see the following output in our terminal:

```
Migrations for 'backoffice':
backoffice/migrations/0001_initial.py
+ Create model Category
+ Create model Dish
+ Create model Order
+ Create model OrderItem
+ Create model Restaurant
+ Add field restaurant to order
+ Add field restaurant to category
```

And to effectively create our tables in the database, we need to run the following command:

```
python manage.py migrate
```

Remember *Migrations are Django’s way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema.*

Registering Our Models in the Django Admin Interface

Now that we have our models created, we need to register them in the Django Admin interface. It’s up to us to decide if we want to see the models in the Django Admin interface or not. For each model registered, the Django Admin interface will automatically create pages which will help us to manage CRUD operations on the model.

To register models in the **Django Admin** interface, we need to edit the file called `admin.py` in our `backoffice` folder.

Django provides us two ways to register a model in the Django Admin interface.

Using ModelAdmin Class

The **class-based** approach can be more explicit for beginners. Let’s see an example of how we can use it with the `Restaurant` model:

```
from django.contrib import admin
from .models import Restaurant

class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('name', 'address', 'phone')
    search_fields = ['name']

admin.site.register(Restaurant, RestaurantAdmin)
```

We created a class `RestaurantAdmin` that inherits from `admin.ModelAdmin`.

We then defined the `list_display` attribute to specify which fields we want to display in the list view. This view is the default view when we click on a model. It is a list of all the objects in the database for that model. We will learn more about the different views in the admin interface in the next chapter.

We also added a `search_fields` attribute to enable searching by name.

Finally, we register our Restaurant model with the `admin.site.register` function, passing in the `RestaurantAdmin` class as the second argument.

With `RestaurantAdmin`, we're telling Django to use that class to control the admin interface for Restaurant—the list view (columns, sorting), search, filtering, and later even the add/edit form layout and behavior.

🔗 By convention, we are adding the keyword `Admin` to our model name to create the admin class name.

Using `@register` decorator

The decorator syntax was introduced in later Django version as a more elegant way to achieve the registration. It's more concise and modern. We will use this syntax in this book.

Let's see how to use it with our Restaurant model:

```
from django.contrib import admin
from .models import Restaurant

@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('name', 'address', 'phone')
    search_fields = ['name']
```

We can see it's quite more easy and rapid. We just need to add the `@admin.register(Restaurant)` decorator above our `RestaurantAdmin` class.

Anyway, both methods will have the same result.

If we go back to our browser on `http://127.0.0.1:8000/admin/` and refresh the page, we will see the result shown in Figure 2-1.



Figure 2-1. Django Admin showing the Backoffice section with the Restaurant model

We now have a Backoffice section in our admin interface with the Restaurant model. If we click the Restaurant link, we will see the list of our restaurants (which is empty for now as shown in Figure 2-2).

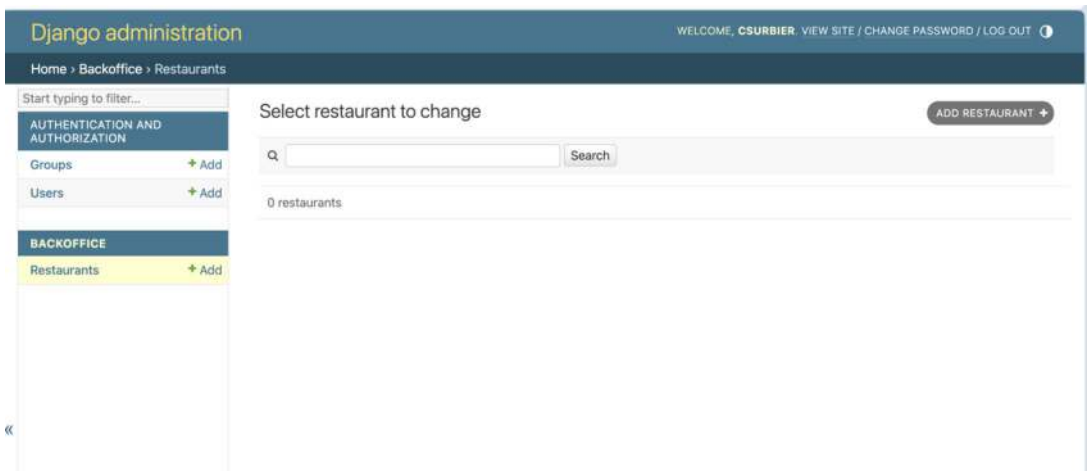


Figure 2-2. Restaurant changelist page (empty list before creating data)

We can also add a new restaurant by clicking the Add restaurant button. Django will automatically create a form for us to fill in the fields as shown in Figure 2-3.

The image shows a screenshot of the Django Admin interface. At the top, the header reads 'Django administration' and 'WELCOME, CSURBIER. VIEW SITE / CHANGE PASSWORD / LOG OUT'. Below the header, a breadcrumb trail shows 'Home > Backoffice > Restaurants > Add restaurant'. On the left sidebar, there are two main sections: 'AUTHENTICATION AND AUTHORIZATION' with links for 'Groups' and 'Users' (each with an '+ Add' link), and 'BACKOFFICE' with a link for 'Restaurants' (also with an '+ Add' link). The main content area is titled 'Add restaurant' and contains several form fields: 'Name:' (a text input), 'Image:' (a file upload button labeled 'Choisir un fichier' and the text 'Aucun fichier choisi'), 'Address:' (a large text area), 'Zip code:', 'City:', 'Country:', and 'Phone:' (all text inputs), and 'Owner:' (a dropdown menu with a '+' icon, a '-' icon, and an 'eye' icon). Below these fields is a checkbox labeled 'Active' which is checked. At the bottom of the form are three buttons: 'SAVE', 'Save and add another', and 'Save and continue editing'.

Figure 2-3. Add restaurant form generated by Django Admin

Creating Fake Data

Before we can start learning on how to use the Django Admin interface, we need to create some fake data to populate our database.

So let's do it; we are going to create some fake data. It will help us to test our application.

To do so, we will use the following libraries:

- Model Bakery (<https://model-bakery.readthedocs.io/en/latest/>)
- Faker (<https://model-bakery.readthedocs.io/en/latest/>)

Model Bakery is a powerful library that helps create test data for Django models. It automatically handles model relationships and can generate appropriate data for different field types. Key features include

- Automatic handling of required fields
- Support for model relationships (ForeignKey, ManyToManyField)
- Custom data generation through recipes

Faker is a Python package that generates realistic fake data including

- Names, addresses, phone numbers
- Company names and job titles
- Dates and times
- Email addresses and usernames
- Geographic data
- And much more

Using the combination of both is super useful when developing and testing admin interfaces. It allows us to quickly generate large amounts of realistic data.

Let's install them using the following command:

```
pip install model_bakery
pip install Faker
```

or add it to our requirements.txt file and install it by running the following command:

```
pip install -r requirements.txt
```

Now let's write a little python script named **createFakeData.py** to generate random data inside our.djangoadmin folder:

```
import os
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "djangoadmin.settings")
import django
django.setup()

from faker import factory, Faker
from backoffice.models import Restaurant, Category, Dish
from model_bakery.recipe import Recipe, foreign_key
from django.contrib.auth.models import User
fake = Faker()
```

CHAPTER 2 CORE CONCEPTS

```
# create some fake User owners
users = Recipe(User,
                username=lambda: fake.user_name(),
                password=lambda: fake.password()
                )

# Create some Restaurants
restaurants = Recipe(Restaurant,
                    name=lambda: fake.company(),
                    address=lambda: fake.address(),
                    zip_code=lambda: fake.postcode(),
                    city=lambda: fake.city(),
                    country=lambda: fake.country,
                    phone=lambda: fake.phone_number(),
                    owner=foreign_key(users))

# Create some Categories
categories = Recipe(Category,
                    name=lambda: fake.word(),
                    restaurant=foreign_key(restaurants))

# Create some Dishes
dishes = Recipe(Dish,
                name=lambda: fake.word(),
                description=lambda: fake.text(),
                price=lambda: fake.random_int(min=1, max=100),
                category=foreign_key(categories))

# Create 10 fake users first
users_list = [users.make() for _ in range(10)]

# Create 2 restaurants for each user
for user in users_list:
    restaurants_list = [restaurants.make(owner=user) for _ in range(2)]

# Create 10 categories for each restaurant
for restaurant in restaurants_list:
    categories_list = [categories.make(restaurant=restaurant) for _ in
                      range(10)]
```

```
# Create 20 dishes for each category
for category in categories_list:
    [dishes.make(category=category) for _ in range(20)]
```

And run our script by typing

```
python createFakeData.py
```

Now, we should have generated data in our database, which we can easily verify by going back to the Django Admin interface and clicking the Restaurant link.

Figure 2-4 shows the restaurant list once it has been populated with the generated data.

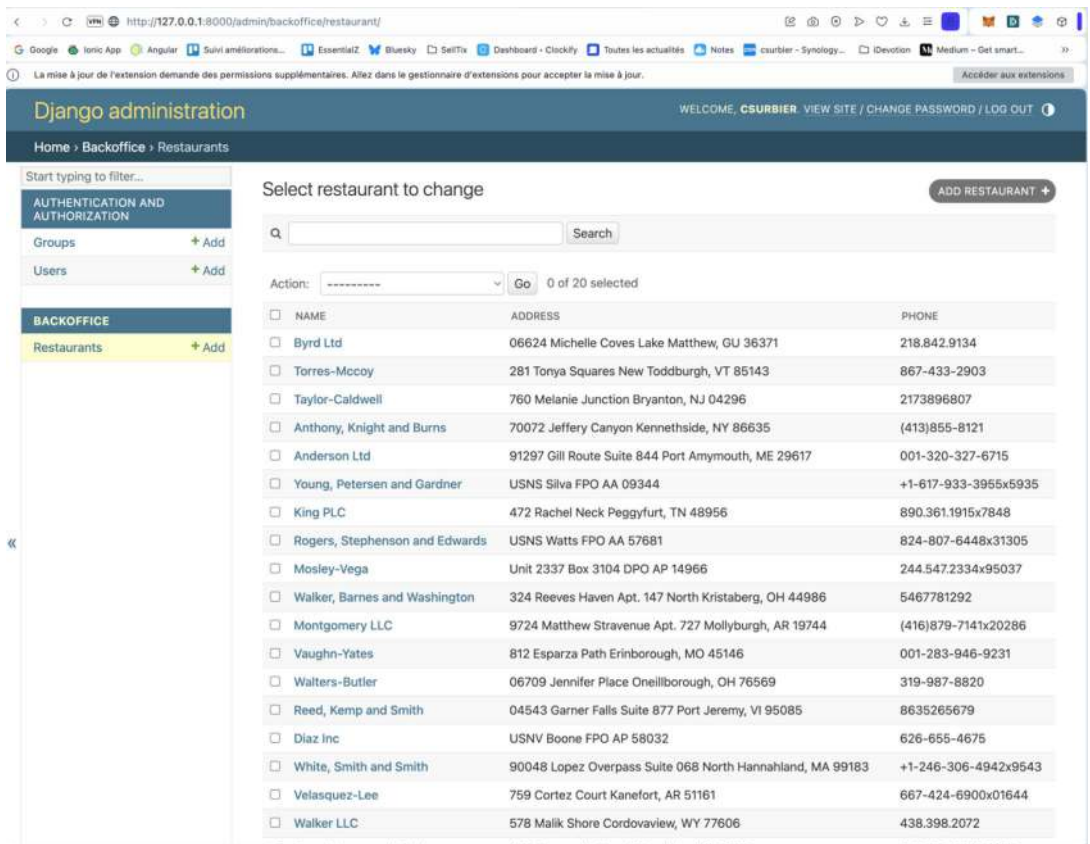


Figure 2-4. Restaurant list populated with generated data and search enabled

As we can see, the **name**, **address**, and **phone** fields (which were defined in our *list_display*) are displayed. We can also use the search field to find a specific restaurant by name.

Please note that since the data are randomly generated, you will not have the same data as the ones shown in the screenshot.

CHAPTER 3

Basic Configuration

Model

The Model class in Django is a fundamental component of the Django framework that represents the structure and behavior of data in a database. It defines the structure of a database table and provides methods for interacting with the data.

The Meta class option in Django models is a powerful tool that allows us to define various model-specific settings that affect how it behaves in the Django Admin interface and elsewhere. Let's focus and review the key Meta options that impact the Django Admin:

```
class Restaurant(models.Model):  
    # ... fields definition ...  
  
    class Meta:  
        verbose_name = 'restaurant' # Used as singular name in admin  
        verbose_name_plural = 'restaurants' # Used as plural name in admin  
        ordering = ['-created_at'] # Default ordering in admin list view  
        unique_together = ['name', 'city'] # Enforces uniqueness in  
        database schema
```

These Meta options affect the Django Admin in the following ways:

1. `verbose_name` and `verbose_name_plural`: Define how the model is displayed in the admin interface.
2. `ordering`: Controls the default ordering of records in the admin list view.
3. `unique_together`: Enforces field uniqueness constraints at database level (by creating an index) and then on top of that in our admin forms. It ensures that the combination of name and city fields must be unique (otherwise, an error will be raised to the user).

Admin

Let's continue by registering other models in our `admin.py` file:

```
from django.contrib import admin
from .models import Restaurant, Category, Dish, Order, OrderItem

@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('name', 'address', 'phone')
    search_fields = ['name']

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ('name', 'restaurant', 'is_active')
    list_filter = ('is_active', 'restaurant')
    search_fields = ('name',)

@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'price', 'is_available')
    list_filter = ('is_available', 'category')
    search_fields = ('name', 'description')

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ('id', 'customer', 'restaurant', 'status', 'total_
amount', 'created_at')
    list_filter = ('status', 'created_at')
    search_fields = ('customer__username', 'restaurant__name')

@admin.register(OrderItem)
class OrderItemAdmin(admin.ModelAdmin):
    list_display = ('order', 'dish', 'quantity', 'unit_price')
    list_filter = ('order__status',)
    search_fields = ('dish__name',)
```

Now we can see all models in the admin interface as shown in Figure 3-1.



Figure 3-1. Django Admin showing the registered Backoffice models (Restaurant, Category, Dish, Order, OrderItem)

If you click Categories or Dishes, you will see some of our randomly generated data.

Customizing the Admin Interface

Let's make some changes to our admin interface by adding the following code in our `admin.py` file:

```
admin.site.site_header = "Uber Eat like Admin"
admin.site.index_title = "Welcome back to Uber Eat like Admin Portal"
```

Reload the page and you will see Figure 3-2.

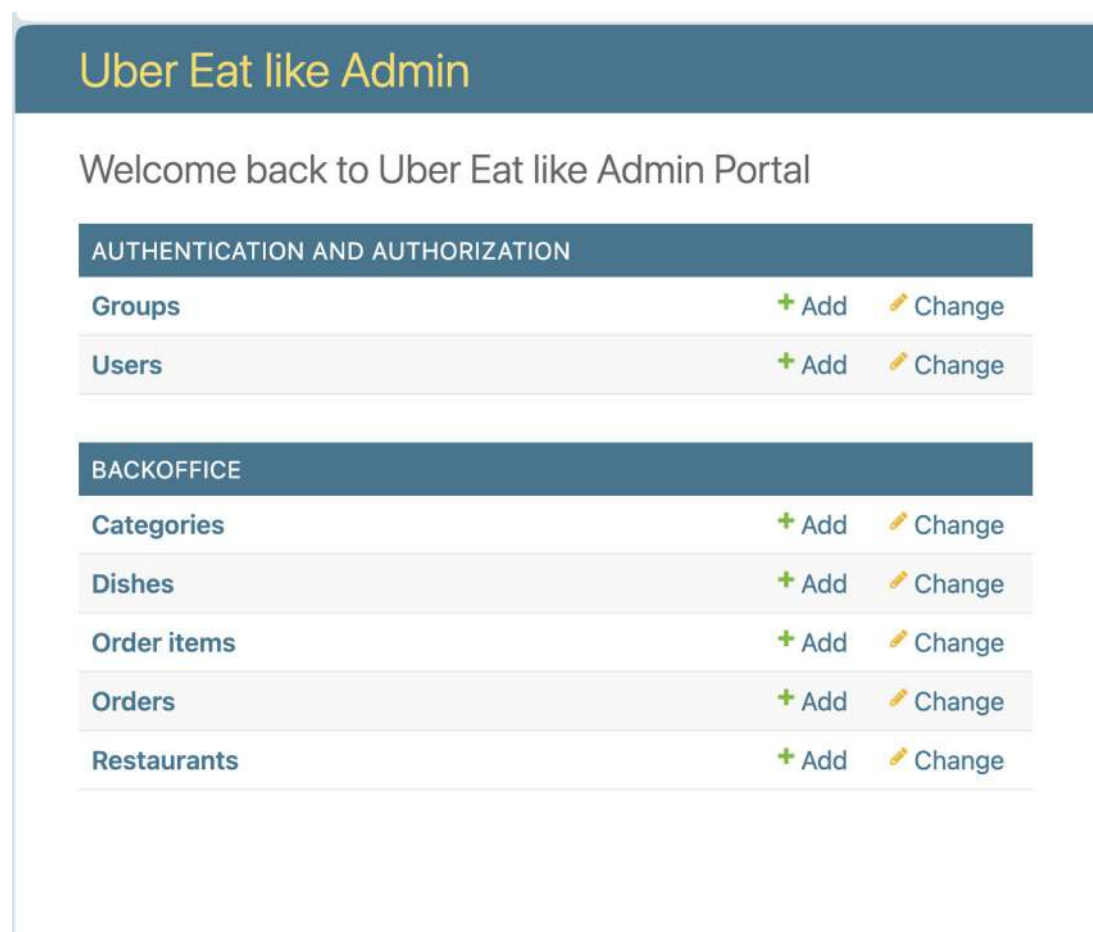


Figure 3-2. Customized admin header and index title

We have change the default admin title and index title with a more accurate definition.

If we pay attention, we can notice a View site link in the top right corner, as shown in Figure 3-3. This link will take us outside the Django Admin site to our main website located at the URL `http://127.0.0.1:8000/`.



Figure 3-3. Default “View site” link displayed in the admin header

But since we are not going to develop a frontend and just focus on the Admin interface, we will remove this link as shown on Figure 3-4 by adding the following code in our `admin.py` file:

```
admin.site.site_url = None # Remove "View Site" link
```



Figure 3-4. Admin header after removing the “View site” link

🐘 If you want to redirect to another URL, just type the URL in the `site_url` variable like this:

```
admin.site.site_url = 'https://www.google.fr' # New link for "View Site"
```

Last customizing for now, we can also add (displayed by default) or remove the `nav_side_bar` for large screen devices by adding the following code in our `admin.py` file:

```
admin.site.enable_nav_sidebar = True # Enable the navigation sidebar
```

Internationalization

By default, Django Admin uses the English language. If we want to support other languages, we must add the `i18n` in our `settings.py` file:

```
MIDDLEWARE = [
    'django.middleware.locale.LocaleMiddleware',
    # other middleware...
]

USE_I18N = True
LANGUAGES = [
    ('en', 'English'),
    ('fr', 'French'),
    ('es', 'Spanish'),
]
```

We also need to specify in our settings.py file, the path where translation files will be stored:

```
import os
LOCALE_PATHS = [
    os.path.join(BASE_DIR, 'locale'),
]
```

The Django Admin interface will automatically appear in the language of the user's browser. If the user's browser language is not supported, Django will use the default language specified in the settings.py file. You can specify the default language in the settings.py file like this:

```
LANGUAGE_CODE = 'fr-FR' # For French
```

Then we need to identify each label we want to translate in our models.py file and use the *specific character to tag them as translatable fields*. Please pay attention to the character that we used in our models.py file, which comes from the django.utils.translation module:

```
from django.utils.translation import gettext_lazy as _ # Use this to get
translated strings
```

```
class Restaurant(models.Model):
    name = models.CharField(_('name'), max_length=200)
    image = models.ImageField(upload_to='restaurants/', blank=True, null=True)
    address = models.TextField(_('address'))
    zip_code = models.CharField(_('zip code'), max_length=20)
    city = models.CharField(_('city'), max_length=100)
    country = models.CharField(_('country'), max_length=100)
    phone = models.CharField(_('phone'), max_length=20)
    owner = models.ForeignKey(User, on_delete=models.PROTECT, related_
name='restaurants')
    is_active = models.BooleanField(_('active'), default=True)
    created_at = models.DateTimeField(auto_now_add=True)

class Meta:
    verbose_name = _('restaurant')
```

```

verbose_name_plural = _('restaurants')
ordering = ['-created_at']

def __str__(self):
    return self.name

```

For the Restaurant model, we have translated the following fields: name, address, zip_code, city, country, phone, and is_active.

Now, if we want a French translation, we need to run the command

```
python manage.py makemessages -l fr
```

It will generate a file named django.po in the locale/fr/LC_MESSAGES/ folder. This file will contain each string to translate:

```

#: backoffice/models.py:6 backoffice/models.py:26 backoffice/models.py:40
msgid "name"
msgstr ""

#: backoffice/models.py:8
msgid "address"
msgstr ""

#: backoffice/models.py:9
msgid "zip code"
msgstr ""

#: backoffice/models.py:11
msgid "country"
msgstr ""

#: backoffice/models.py:12
msgid "phone"
msgstr ""

```

We need to edit the msgstr field with the French translation:

```

#: backoffice/models.py:6 backoffice/models.py:26 backoffice/models.py:40
msgid "name"
msgstr "nom"

#: backoffice/models.py:8

```

```
msgid "address"
msgstr "adresse"

#: backoffice/models.py:9
msgid "zip code"
msgstr "code postal"

#: backoffice/models.py:11
msgid "country"
msgstr "pays"

#: backoffice/models.py:12
msgid "phone"
msgstr "téléphone"
```

Then we need to compile the translation files:

```
python manage.py compilemessages
```

If we reload our admin interface, as shown in Figure 3-5, we can see that the default Django Admin interface and labels are now in French.



Figure 3-5. Django Admin interface translated to French

🐭 If you make a translation change in the file, you need to recompile the translation files to see it.

To know more about translation possibilities in Django, please refer to the [Django documentation](https://docs.djangoproject.com/en/5.1/topics/i18n/translation/) (<https://docs.djangoproject.com/en/5.1/topics/i18n/translation/>). It contains a lot of useful information and in particular how to format dates, times, and numbers for different languages.

Summary

In this chapter, we set up the foundations for working with Django Admin. We created a project and app, defined models, ran migrations, and logged into the admin so we could manage real objects from the database. We also explored how Django Admin discovers models, how registration works, and why small customizations (like list columns and search) can quickly improve usability.

Finally, we introduced internationalization as an option you can enable when you need it. With these basics in place, we're ready to go deeper into the essential features that make Django Admin such a powerful backoffice.

PART II

Essential Features

In this part, we'll dive deep into Django Admin's core functionalities that transform it from a basic CRUD interface into a powerful business operations platform. We'll explore

Display and Filtering: Learn how to create intuitive list views with advanced filtering capabilities. We will master techniques for presenting data effectively using `list_display` options, implement sophisticated filter systems, and build search functionality.

Actions and Operations: Discover how to extend Django Admin beyond simple data management by implementing custom actions. We'll cover both built-in actions and creating custom operations that can process multiple records simultaneously.

Forms and Validation: Take control of data input by customizing forms and implementing robust validation rules.

Inline Editing: Managing related models efficiently using inline editing capabilities. We'll compare tabular and stacked inlines.

Performance Optimization: Ensure our admin interface remains fast and responsive even with large datasets. We'll cover essential query optimization techniques, implement caching strategies, and manage pagination for optimal performance.

By the end of this part, you'll have a comprehensive understanding of Django Admin's essential features and be able to implement them effectively in your projects.

CHAPTER 4

Display and Filtering

List Display Options

In Chapter 1, we have seen how to create a model, register it to the admin, and display it on the admin interface. The `list_display` attribute in Django Admin controls which fields appear as columns in the list view. Let's look at how we define it:

```
from django.contrib import admin
from .models import Restaurant

@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('name', 'address', 'phone')
```

The `list_display` accepts a tuple of model field names as strings. Each field specified will

- Create a column in the admin list view
- Display the corresponding model field value
- Maintain the order specified in the tuple

For example, if we wanted to show more fields:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('name', 'address', 'phone', 'city', 'is_active',
                    'created_at')
    search_fields = ['name']
```

This would create six columns in the admin interface, displaying each restaurant's name, address, phone number, city, active status, and creation date as shown in Figure 4-1.

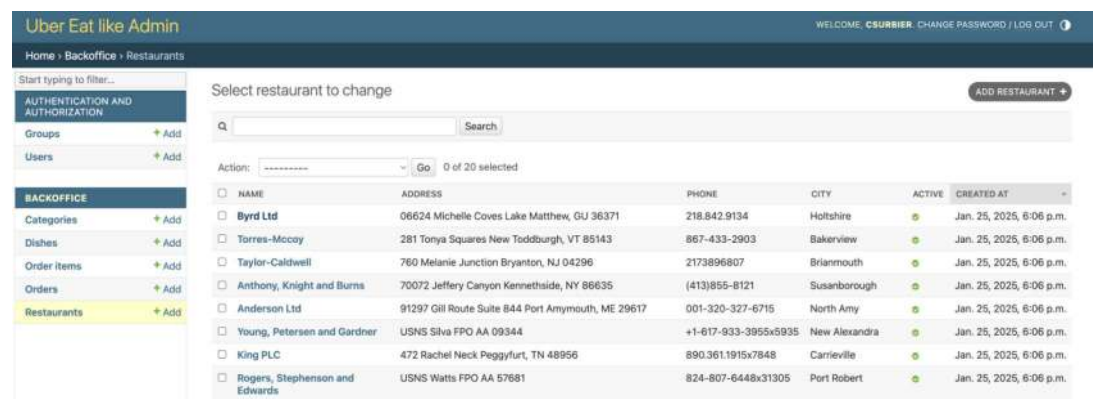


Figure 4-1. Restaurant changelist with extended `list_display` columns (city, active status, and created date)

We can notice that the `is_active` field is a boolean field, so it is displayed as a checkbox. It's a default **Django Admin** feature.

Date

We can add a header above the `list_view` to display a navigation bar based on date as shown on Figure 4-2. To do so, we have to add the `date_hierarchy` attribute to the `admin.ModelAdmin` class with the date field name we want to navigate by.

```
date_hierarchy = 'created_at'
```

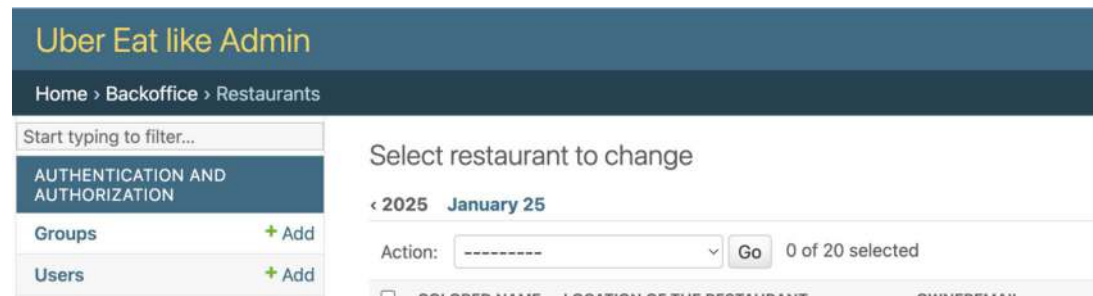


Figure 4-2. `date_hierarchy` navigation bar for browsing records by creation date

We also have the ability to customize how fields are displayed on the interface. For now, we have used and displayed the default field name. But we can go much more beyond that.

Customizing Field Names

- Using a custom method:

```
@admin.register(Restaurant)

class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address")
    def full_address(self, obj):
        return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"
    list_display = ('name', 'full_address', 'phone', 'city', 'is_active', 'created_at')
```

We have created a method **full_address** which returns a string containing the address, zip code, and city. We have used the `@admin.display` decorator to customize the field name. The `description` argument is used to set the field name which will appear as column name (see Figure 4-3) in the admin interface, Full address in our case.

NAME	FULL ADDRESS	PHONE	CITY	ACTIVE	CREATED AT
Byrd Ltd	06624 Michelle Coves Lake Matthew, GU 36371 zip code: 12775 City: Holtshire	218.842.9134	Holtshire	●	Jan. 25, 2025, 6:06 p.m.
Torres-Mecoy	281 Tonya Squares New Toddburgh, VT 85143 zip code: 34877 City: Bakerview	867-433-2903	Bakerview	●	Jan. 25, 2025, 6:06 p.m.
Taylor-Caldwell	760 Melanie Junction Bryantton, NJ 04296 zip code: 03161 City: Brianmouth	2173896807	Brianmouth	●	Jan. 25, 2025, 6:06 p.m.
Anthony, Knight and Burns	70072 Jeffery Canyon Kennettside, NY 86635 zip code: 23730 City: Susanborough	(413)855-8121	Susanborough	●	Jan. 25, 2025, 6:06 p.m.
Anderson Ltd	91297 Gill Route Suite 844 Port Amymouth, ME 29617 zip code: 07880 City: North Amy	001-320-327-6715	North Amy	●	Jan. 25, 2025, 6:06 p.m.

Figure 4-3. Custom `list_display` column using an admin method with a custom header (“Full address”)

- Using a custom method in the **model** class:

We can also define a method in our model class (models.py file) and then use it in the `list_display` attribute. Let's edit our models.py file and modify our Restaurant model:

```
from django.contrib import admin

class Restaurant(models.Model):
    name = models.CharField(_('name'), max_length=200)
    image = models.ImageField(upload_to='restaurants/',
                              blank=True, null=True)
    address = models.TextField(_('address'))
    zip_code = models.CharField(_('zip code'), max_length=20)
    city = models.CharField(_('city'), max_length=100)
    country = models.CharField(_('country'), max_length=100)
    phone = models.CharField(_('phone'), max_length=20)
    owner = models.ForeignKey(User, on_delete=models.PROTECT,
                              related_name='restaurants')
    is_active = models.BooleanField(_('active'), default=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        verbose_name = _('restaurant')
        verbose_name_plural = _('restaurants')
        ordering = ['-created_at']

    def __str__(self):
        return self.name

    @admin.display(description="A custom field")
    def custom_field(self):
        return f"My custom string"
```

Now we can use the `custom_field` method in the `list_display` attribute of our admin class:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address")
```

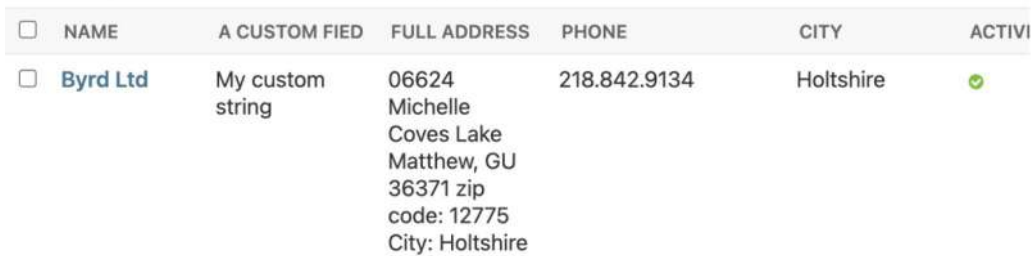
```

def full_address(self,obj):
    return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

list_display = ('name', 'custom_field','full_address',
                'phone', 'city', 'is_active', 'created_at')

```

And we can see our new column added in the interface as shown on Figure 4-4:



☐	NAME	A CUSTOM FIED	FULL ADDRESS	PHONE	CITY	ACTIVI
☐	Byrd Ltd	My custom string	06624 Michelle Coves Lake Matthew, GU 36371 zip code: 12775 City: Holtshire	218.842.9134	Holtshire	

Figure 4-4. *list_display showing a custom method defined on the model*

- Using a **@property** in the **model** class:

This method is quite similar to the previous one. We will define a property in our model class and use it in the `list_display` attribute of our admin class.

```

@property
@admin.display(
    description="Location of the restaurant",
    boolean=False,
)
def location(self):
    return self.city + " " + self.country

```

and can use it in the `list_display` attribute of our admin class:

```

list_display = ('name','location','ownerEmail','full_address',
                'phone', 'city', 'is_active', 'created_at')

```

🚩 In `@admin.display(...)`, the `boolean=` option tells Django Admin whether this method/property should be rendered as a boolean icon column (a green check/red cross UI).

Now let's try something more complex!

We will create a method to display the name of the Restaurant in red if the restaurant is not active; otherwise, it will be displayed in green. To do so, instead of return a string, we will return a string with an **html** code. To do that, there is a special method named **format_html**. Let's import it and use it in our `models.py` file:

```
from django.utils.html import format_html
```

and now let's create our customized method:

```
@admin.display
def colored_name(self):
    if self.is_active:
        color_code = '00FF00' # Green color code
    else:
        color_code = 'FF0000' # Red color code
    return format_html(
        '<span style="color: #{};">{</span>',
        color_code,
        self.name
    )
```

If our Restaurant is active, the name will be displayed in green; otherwise, it will be displayed in red.

We can use this method instead of our name field in our `list_display` attribute:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address")
    def full_address(self, obj):
        return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

    list_display = ('colored_name', 'custom_field', 'full_address', 'phone',
                   'city', 'is_active', 'created_at')
```

And voila, see Figure 4-5.

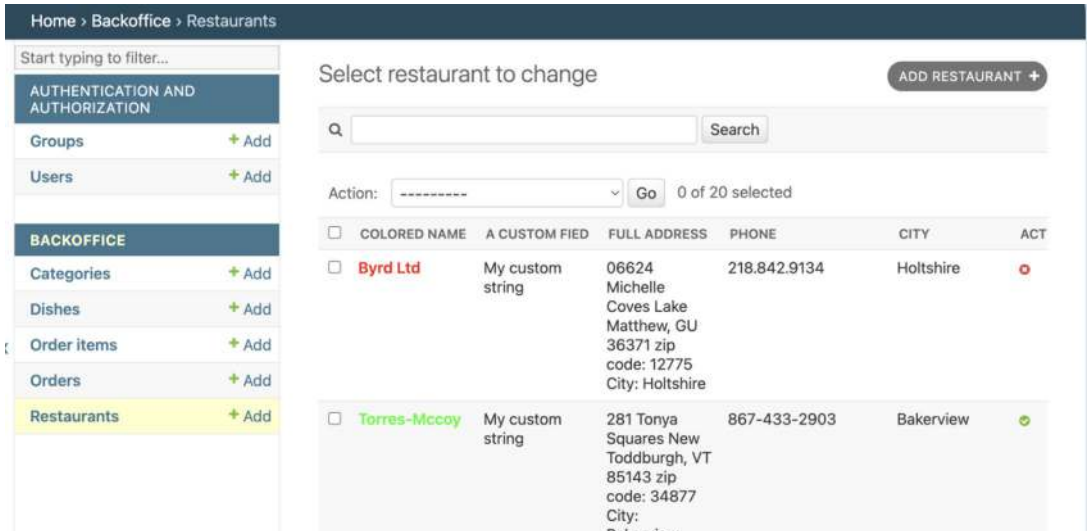


Figure 4-5. Custom HTML rendering in `list_display` to color a restaurant name by active status

🔗 The `format_html` method is used to format a string as HTML. It is a good practice to use it in the admin instead of in the `models.py` file.

Displaying a Field from a Related Model

Our `Category` has a `restaurant` field which is a foreign key to the `Restaurant` model. Let's have a look to our `CategoryAdmin` definition:

```
@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):
    list_display = ('name', 'restaurant', 'is_active')
    list_filter = ('is_active', 'restaurant')
```

The `list_display` contains the `restaurant` field which will display the restaurant name value as we can see in Figure 4-6.

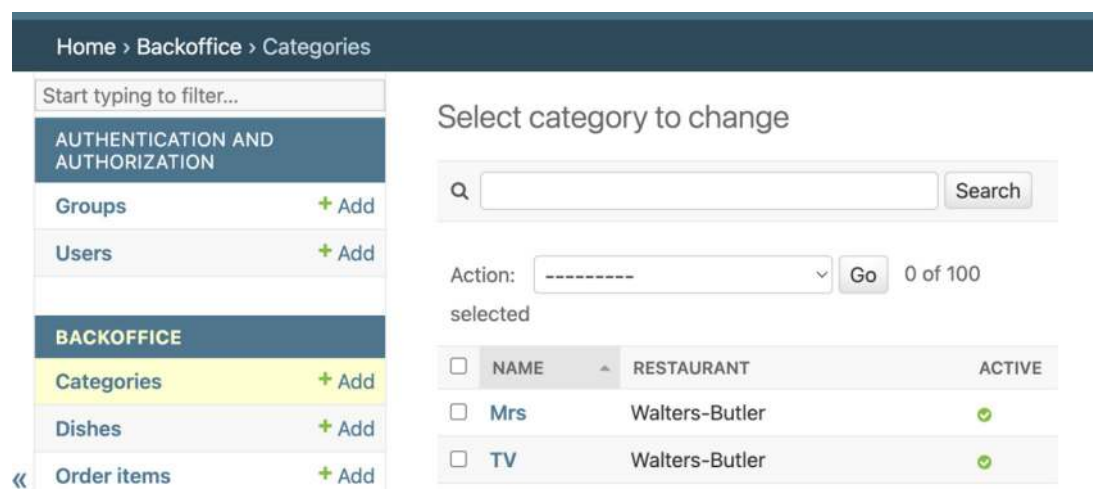


Figure 4-6. *Displaying a related model value in the list view via the `__str__` method (Category shows its Restaurant)*

The value displayed comes from the `__str__` method of the Restaurant model:

```
def __str__(self):
    return self.name
```

So we can change this method to adapt to our needs:

```
def __str__(self):
    return f"{self.name} from {self.city}"
```

which will display the name and the city of the restaurant (Figure 4-7).

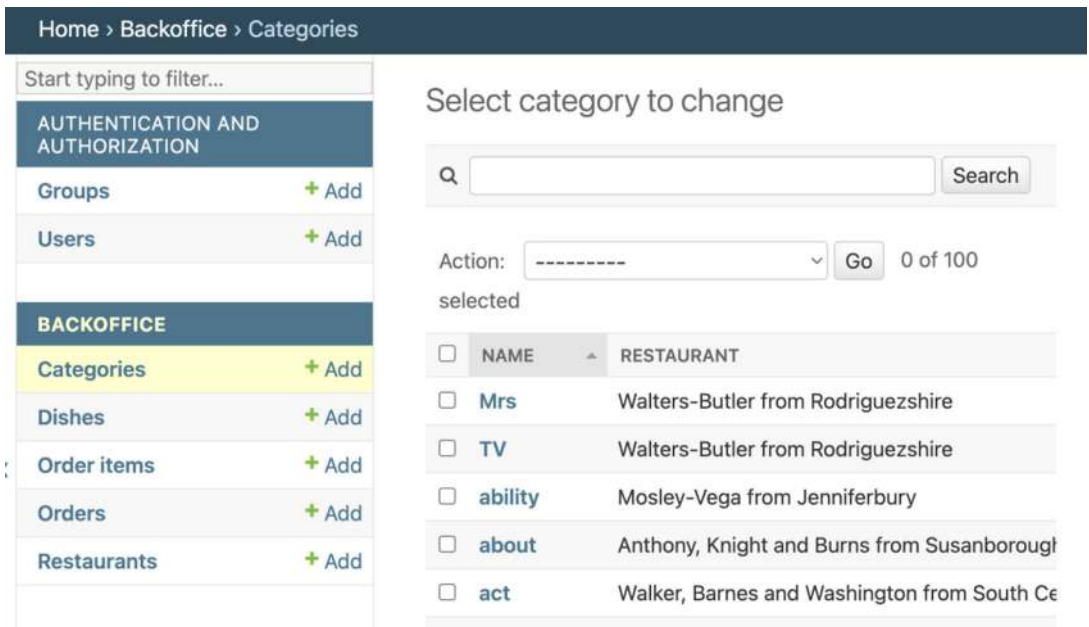


Figure 4-7. Updated `__str__` output displayed in the admin list (restaurant name with its city)

Let's go back to our Restaurant model which has a ForeignKey to the User model, which is owner field. It could be nice to display the **email** of that foreign key object (see Figure 4-8). To do that, we can create a custom method **get_owner_email** in our RestaurantAdmin class:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address")
    def full_address(self, obj):
        return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

    @admin.display(description='Owner email')
    def get_owner_email(self, obj):
        return obj.owner.email

    list_display = ('colored_name', 'get_owner_email', 'full_address',
                    'phone', 'city', 'is_active', 'created_at')
```

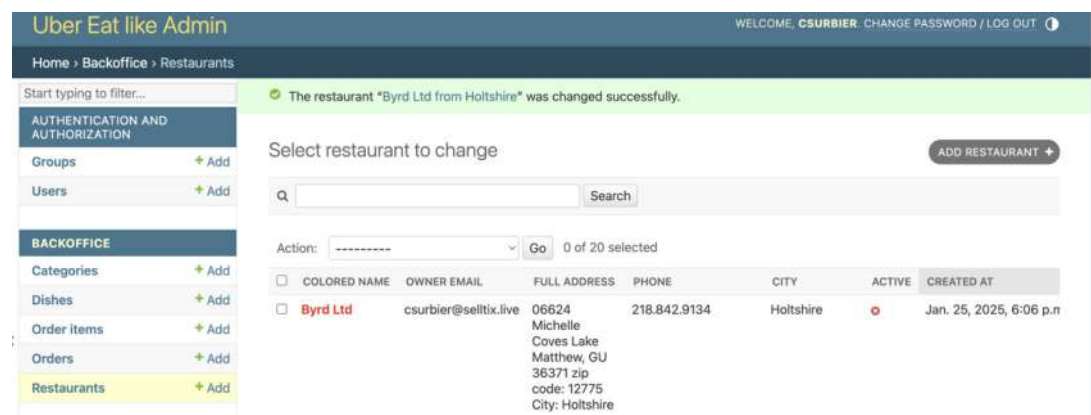


Figure 4-8. Restaurant list including a related-field column for the owner email

Or, starting with **Django 5.1**, we can display fields from related models directly in `list_display` using Django’s double-underscore (`__`) syntax (e.g., `owner__email`), so we don’t need a custom method:

```
list_display = ('colored_name', 'owner__email', 'full_address', 'phone',
'city', 'is_active', 'created_at')
```

list_per_page

By default, Django Admin paginates result in the interface by 100. We can change this behavior by setting a **list_per_page** value (`list_per_page = 3`). Figure 4-9 shows result with the pagination controls.

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address")
    def full_address(self, obj):
        return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

    list_display = ('colored_name', 'owner__email', 'full_address', 'phone',
'city', 'is_active', 'created_at')
    list_per_page = 3
```

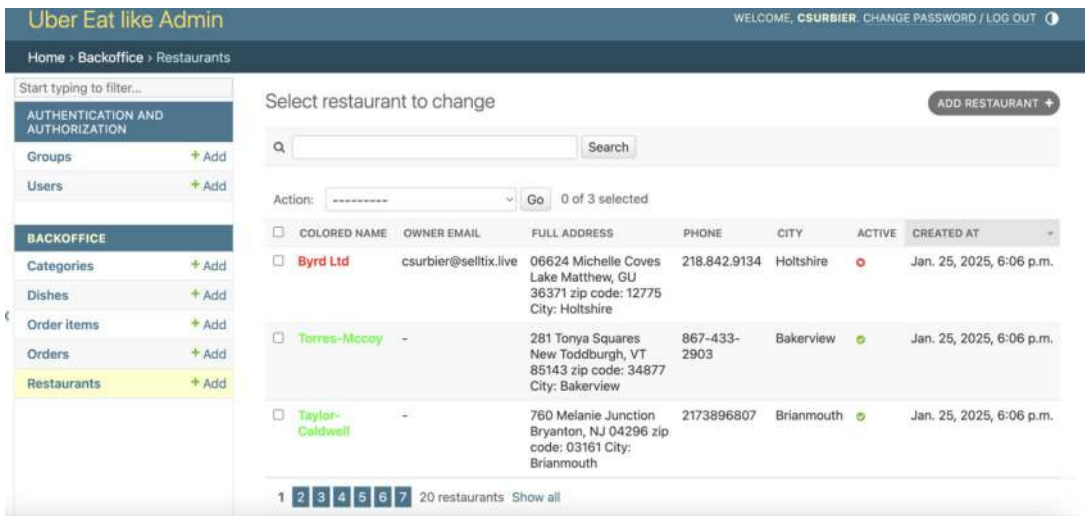



Figure 4-9. *Pagination controls after setting `list_per_page` to a small value*

And now we have a pagination of three items per page, and we can see the pagination buttons in the bottom of the page.

If we pay attention to this screenshot results, we can see that some owners don't have an email, and so by default, Django Admin displays a '-' character. To change this default behavior, we can use the `empty_value_display` attribute in our `RestaurantAdmin` to indicate the value we want to display when the value is empty:

```
empty_value_display = "-empty-"
```

Ordering Instruction

We can order results in the list view interface, by clicking on the column name. You can try by clicking on the phone or city columns and see the difference.

However, for custom methods added, the ordering is not working. To fix this, we can use the `ordering` attribute in the `@admin.display` decorator in our `admin.py` file:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address", ordering="address")
    def full_address(self, obj):
```

```

return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

list_display = ('colored_name', 'owner__email', 'full_address', 'phone',
               'city', 'is_active', 'created_at')
search_fields = ['name']
list_per_page = 100

```

Now the `full_address` method is ordered by the address field and is clickable on the admin interface.

Or to order by the name field for a custom `colored_name` method, in our `models.py` file:

```

@admin.display(ordering="name")
def colored_name(self):
    if self.is_active:
        color_code = '00FF00' # Green
    else:
        color_code = 'FF0000' # Red
    return format_html(
        '<span style="color: #{};">{}</span>',
        color_code,
        self.name
    )

```

To indicate descending order with the `ordering` argument, you can use a hyphen prefix on the field name like this:

```

@admin.display(ordering="-name")
def colored_name(self):
    if self.is_active:
        color_code = '00FF00' # Green
    else:
        color_code = 'FF0000' # Red
    return format_html(
        '<span style="color: #{};">{}</span>',
        color_code,
        self.name
    )

```

The ordering argument supports query lookups to sort by values on related models. Meaning, to order by the email of the owner of the restaurant, we can use the `owner__email` as ordering argument. Let's write a method as example:

```
@admin.display(ordering="owner__email")
def ownerEmail(self, obj):
    return obj.owner.email

list_display = ('colored_name', 'ownerEmail', 'full_address', 'phone',
               'city', 'is_active', 'created_at')
```

If you need ordering based on a computed value, annotate it onto the `changelist_queryset` (typically in `ModelAdmin.get_queryset()`), then reference that annotation in `@admin.display(ordering="...")`. We'll cover this in Chapter 3.

list_display_links

If you paid attention to the Restaurant list, you should have notice that by clicking on the first column (colored name), we can access the change view of the restaurant (as shown on Figure 4-10).

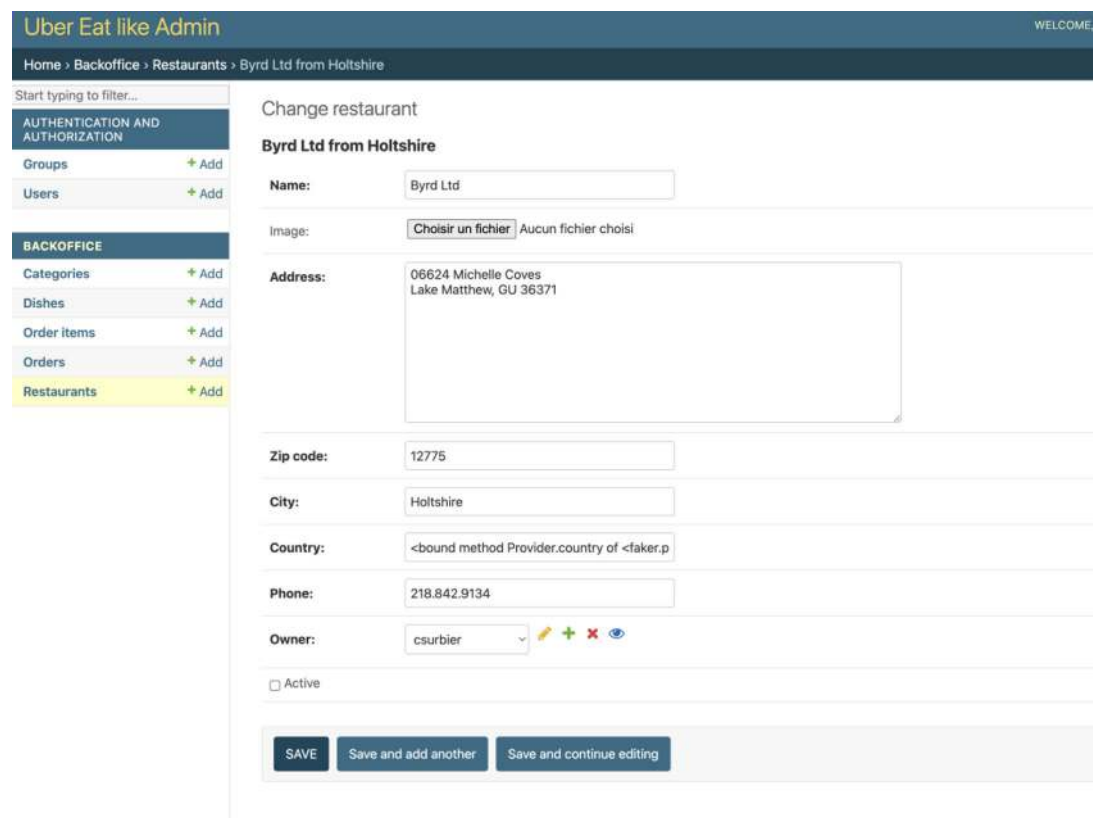


Figure 4-10. Default clickable column behavior in the changelist (`list_display_links`)

By default, the list view of Django Admin use the first column item as a link to the change view of that item. If we want to change the default behavior, we can use the `list_display_links` attribute and just specify the fields/column we would like to enable the click:

```
list_display_links = ('colored_name', 'ownerEmail')
```

If we don't want any column to be clickable, we can set the `list_display_links` to `None` value.

list_editable

If we want to edit the value of a field directly from the list view, we can use the `list_editable` attribute:

```
list_editable = ('colored_name',)
```

Because `colored_name` is a custom method, we will have an error: *The value of 'list_editable[0]' refers to 'colored_name', which is not a field of 'backoffice.Restaurant'.*

Let's remove our custom method and use again the name model field in our list display:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address",ordering="address")
    def full_address(self,obj):
        return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

    @admin.display(ordering="owner__email")
    def ownerEmail(self, obj):
        return obj.owner.email

    list_display = ('name','location','ownerEmail','full_address', 'phone',
                    'city', 'is_active', 'created_at')
    list_display_links = ('ownerEmail',)
    list_editable = ('name',)
    list_per_page = 3
```

Now the name field is editable from the list view, and we also have a SAVE button at the bottom of the page (See Figure 4-11).

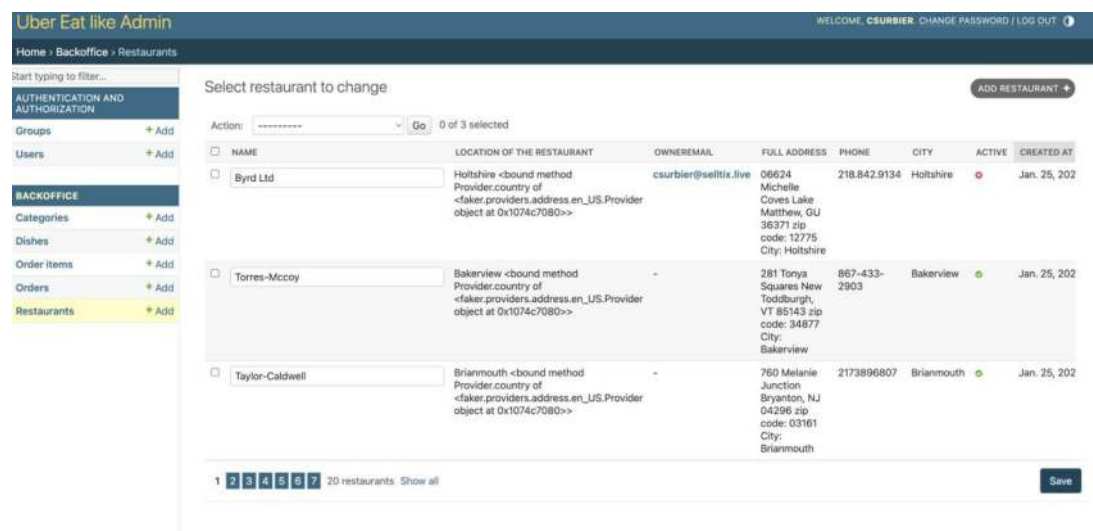


Figure 4-11. *Inline editing enabled in the changelist using `list_editable`*

Warnings The name field can't be in the `list_display_links` and the `list_editable` at the same time. We need to make a choice.

Any field in `list_editable` must also be in `list_display`. You can't edit a field that's not displayed which makes sense.

Filter on the Change List

Django Admin provides powerful filtering capabilities through the `list_filter` attribute. Filters appear in the right sidebar of the list view (as shown on Figure 4-12) and allow users to narrow down the displayed records.

Basic filtering can be implemented by adding the `list_filter` attribute to our `ModelAdmin` class and we just need to pass the fields we want to filter on:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('name', 'city', 'is_active', 'created_at')
    list_filter = ('is_active', 'city', 'created_at')
```

The screenshot shows a Django admin interface for managing restaurants. On the left is a sidebar with navigation links: 'AUTHENTICATION AND AUTHORIZATION' (Groups, Users), 'BACKOFFICE' (Categories, Dishes, Order items, Orders), and 'Restaurants' (highlighted). The main area is titled 'Select restaurant to change' and shows a list of 10 restaurants. A filter sidebar on the right allows filtering by 'active' status, 'city', and 'created at' date. The table columns are: COLORED NAME, LOCATION OF THE RESTAURANT, OWNEREMAIL, FULL ADDRESS, and PHONE.

COLORED NAME	LOCATION OF THE RESTAURANT	OWNEREMAIL	FULL ADDRESS	PHONE
☐ Byrd Ltd	Holtshire <bound method Provider.country of <faker.providers.address.en_US.Provider object at 0x1074c7080>>	csubler@sellitx.live	06624 Michelle Coves Lake Matthew, GU 36371 zip code: 12775 City: Holtshire	218.842.9134
☐ Torres-McCoy	Bakerview <bound method Provider.country of <faker.providers.address.en_US.Provider object at 0x1074c7080>>	-empty-	281 Tonya Squares New Toddburgh, VT 85143 zip code: 34877 City: Bakerview	867-433-2903
☐ Taylor-Caldwell	Brianmouth <bound method Provider.country of <faker.providers.address.en_US.Provider object at 0x1074c7080>>	-empty-	760 Melanie Junction Bryanton, NJ 04296 zip code: 03161 City: Brianmouth	2173896807
☐ Anthony, Knight and Burns	Susanborough <bound method Provider.country of <faker.providers.address.en_US.Provider object at 0x1074c7080>>	-empty-	70072 Jeffery Canyon Kennethside, NY 86635 zip code: 23730 City: Susanborough	(413)855-8121
☐ Anderson Ltd	North Amy <bound method Provider.country of <faker.providers.address.en_US.Provider object at 0x1074c7080>>	-empty-	91297 Gill Route Suite 844 Port Amymouth, ME 29617 zip code: 07880 City: North Amy	001-320-327-6715
☐ Young, Petersen and Gardner	New Alexandra <bound method Provider.country of <faker.providers.address.en_US.Provider object at 0x1074c7080>>	-empty-	USNS Silva FPO AA 09344 zip code: 18320 City: New	+1-617-933-3955x5835

Figure 4-12. Basic sidebar filters created with `list_filter`

As we can see, we now have a filter column on the right side of the list view, and we can filter by `is_active`, `city`, and `created_at`. Based on the choice you made in that list, the displayed records will be filtered.

Please note that if we use a field for filtering that has lot of different values, the list of values in the interface will be too long, and it will be difficult to use. In that case, prefer **search_fields** or a custom filter (see below in the “Custom Filters” or “Search on the Change List” section).

We can also use relationship fields to filter on related models. For example, if we want to filter by the owner of the restaurant, we can use the owner field:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
```

```
list_display = ('name', 'city', 'is_active', 'created_at')
list_filter = ('is_active', 'city', 'created_at', 'owner')
```

In that case, a new filter column will be added to the right side of the list view (see Figure 4-13) with the name value of the owner field.

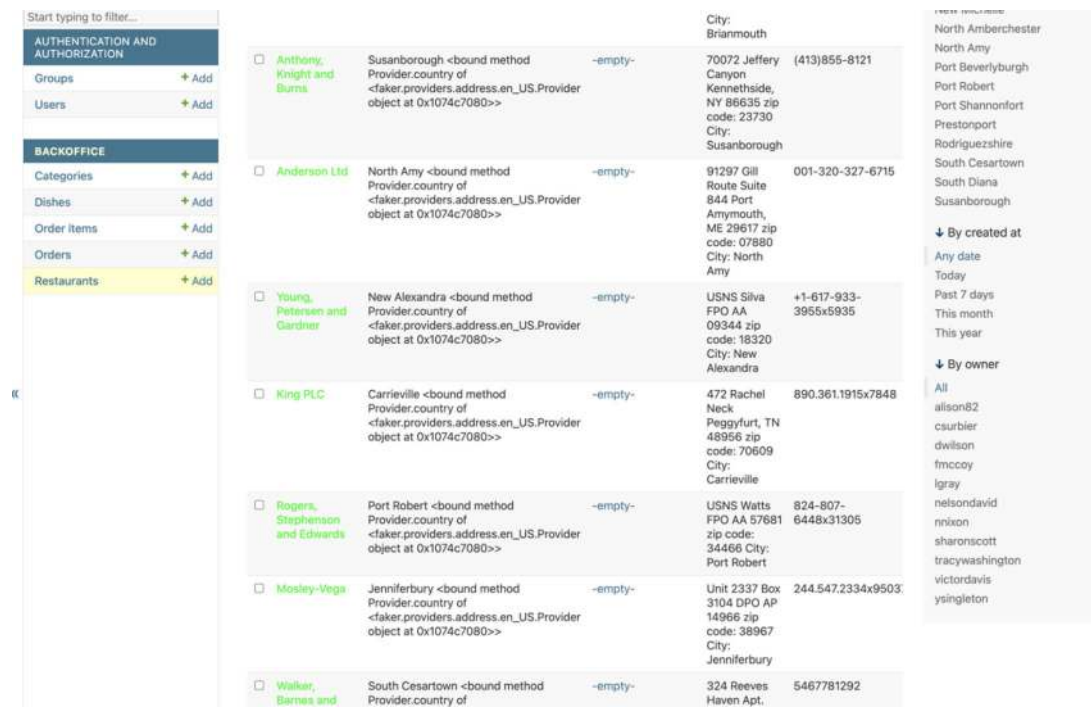


Figure 4-13. Filtering by a related model field (owner) using list_filter

Related Field Filters

We can also filter by fields from related models using double underscores (__) syntax (see Figure 4-14 for results). For example, if we want to filter by the owner’s email, we can use the owner_email field:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('name', 'city', 'is_active', 'created_at')
    list_filter = ('is_active', 'city', 'created_at', 'owner_email')
```


☒ Gardner	<faker.providers.address.en_US.Provider object at 0x1074c7080>		09344 zip code: 18320 City: New Alexandra	
☐ King PLC	Carrieville <bound method Provider.country of <faker.providers.address.en_US.Provider object at 0x1074c7080>	-empty-	472 Rachel Neck Peggyfurt, TN 48956 zip code: 70609 City: Carrieville	890.361.1915x7848
☐ Rosens	Port Robert <bound method	-empty-	LISNS Waits	824-R07-

This year

↓ By email address

All

csurbler@selltic.live

Figure 4-14. Filtering on a related model attribute using double-underscore syntax (`owner__email`)

Custom Filters

Even if Django Admin provides built-in filters, as we just saw, sometimes we need more flexibility, and in that case, we can implement our own custom filters.

To implement our custom filters, we need to extend the **SimpleListFilter** class, and we need to provide the following:

- **title:** The title of the filter which is the column name that will appear in the filter column. **Title is a class attribute (i.e., class-level property/field).** Django reads it to render the sidebar filter label.
- **parameter_name:** The name of the parameter that will be used in the URL to filter the queryset. **Parameter_name is a class attribute (i.e., class-level property/field).** Django reads it to know which GET parameter to look for in the URL.

If you paid attention when clicking on a filter field, **Django Admin** adds a parameter in the URL with the name of the filter and the value of the filter. For example, if we click on the `is_active` filter, the URL will be `http://127.0.0.1:8000/admin/backoffice/restaurant/?is_active=True`:

- **lookups:** A tuple of tuples containing the display name and the value of the filter
- **queryset:** The method that will be called to filter the queryset

lookups and **queryset** are methods we override from the **SimpleListFilter** class. Django calls them at runtime.

Let's have a look at the following example (see Figure 4-15 for result):

```
class RestaurantOwnerFilter(admin.SimpleListFilter):
    title = 'Owner Type' # class attribute
    parameter_name = 'owner_type' # class attribute

    def lookups(self, request, model_admin): # method override from SimpleListFilter class
        return (
            ('with_owner', _('Has Owner')),
            ('without_owner', _('No Owner')),
        )

    def queryset(self, request, queryset): # method override from SimpleListFilter class
        if self.value() == 'with_owner':
            return queryset.exclude(owner=None)
        if self.value() == 'without_owner':
            return queryset.filter(owner=None)
```

`self.value()` is a method that returns the value of the parameter that was selected in the filter.

To use it in our admin class:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_filter = (RestaurantOwnerFilter, 'is_active', 'city')
```

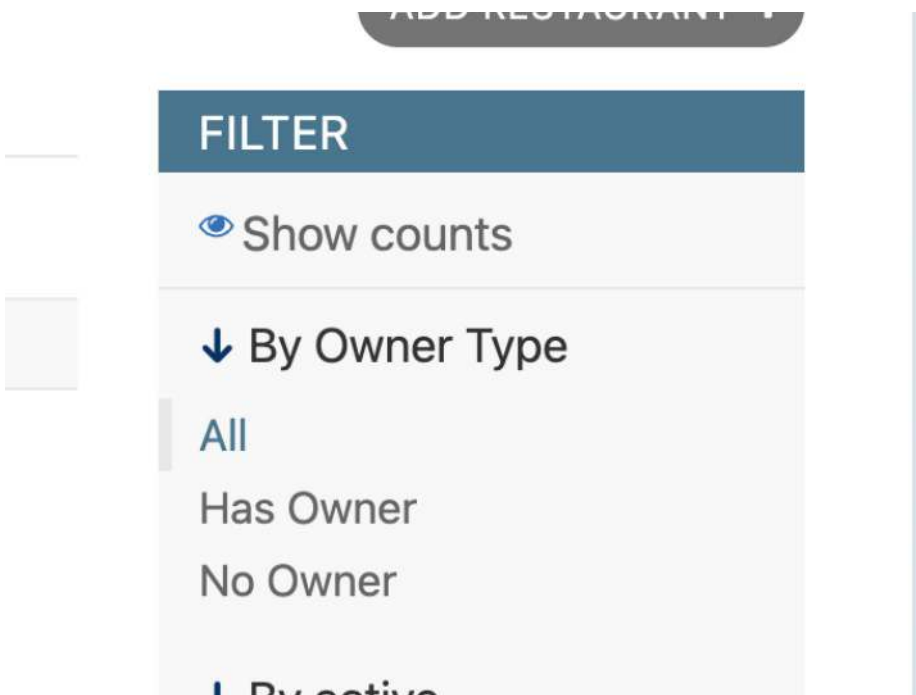


Figure 4-15. Custom `SimpleListFilter` (“Owner Type”) added to the changelist sidebar

Now we have our new Owner Type filter in the list view, and if we click on it, we can see the results of the filter and notice that the URL called is `http://127.0.0.1:8000/admin/backoffice/restaurant/?owner_type=with_owner` or `http://127.0.0.1:8000/admin/backoffice/restaurant/?owner_type=without_owner` which are the values we defined in our `lookups` method.

The **request** object and **model_admin** objects are passed to the **lookups** method, so we can use them to even customize further the filter. Let’s say we want only **superuser** to be able to use our custom filter, we can do the following:

```
def lookups(self, request, model_admin):
    if not request.user.is_superuser:
        return ()
    return (
```

```

        ('with_owner', _('Has Owner')),
        ('without_owner', _('No Owner')),
    )

```

is_superuser is a built-in method that returns True if the user is a superuser (i.e., an admin administrator). If the user is a superuser, we return the tuple of tuples containing the display name and the value of the filter; otherwise, we return an empty tuple, so the filter will not be displayed.

We can use the **model_admin** object in the lookups method; for example, if we want to look at the available data in the restaurant database values before displaying the filter, we can do the following:

```

class RestaurantWithoutPhoneFilter(admin.SimpleListFilter):
    title = 'Restaurant without phone'
    parameter_name = 'phone_filter'

    def lookups(self, request, model_admin):
        qs = model_admin.get_queryset(request)
        if qs.filter(
            phone=''
        ).exists():
            yield ("no_phone", _("No phone set"))

    def queryset(self, request, queryset):
        if self.value() == 'no_phone':
            return queryset.filter(phone='')
        return queryset

```

That way, the filter option “Restaurant without phone” will only be displayed if we have at least one restaurant without a phone number.

```

list_filter = (RestaurantOwnerFilter, RestaurantWithoutPhoneFilter, 'is_
active', 'city',)

```

To test the filter, we can create a restaurant without a phone number and see if the filter is displayed. But before that, we need to update our `models.py` file to add to the phone field the option **blank=True**, which means we can enter an empty phone number in our admin:

```
phone = models.CharField(_('phone'), max_length=20, blank=True)
```

Note This previous code is only for demonstration purposes, because **Django Admin** has a built-in filter to do that—the **admin.EmptyFieldListFilter** which is a filter that will display a filter if the field is empty (as shown in Figure 4-16):

```
list_filter = ( ("phone", admin.EmptyFieldListFilter), RestaurantOwnerFilter, 'is_active', 'city', )
```

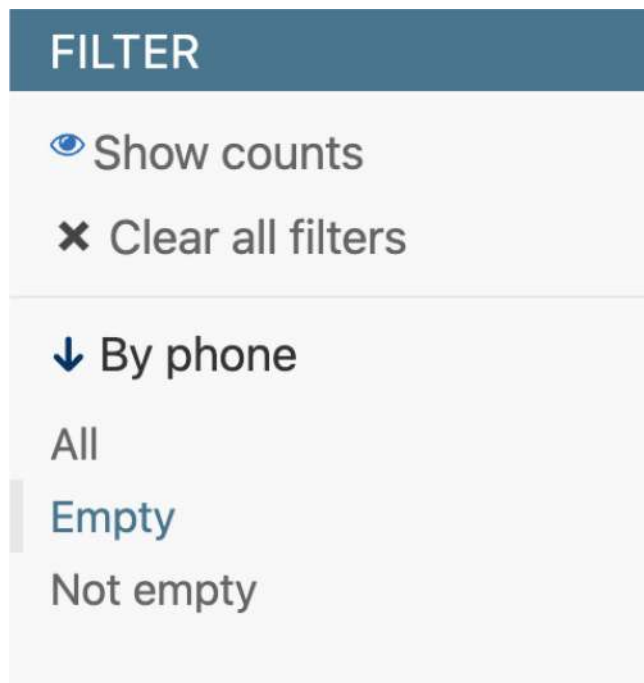


Figure 4-16. Filtering empty values with `admin.EmptyFieldListFilter` (restaurants without a phone)

If you want to go further, check the [official documentation](https://docs.djangoproject.com/en/5.1/ref/contrib/admin/filters/#modeladmin-list-filters) (<https://docs.djangoproject.com/en/5.1/ref/contrib/admin/filters/#modeladmin-list-filters>).

Search the Change List

We have seen how to display items in a list, how to filter them, and how to sort them. Now let’s see how we can search for items.

To search for items in the list view, we can use the `search_fields` attribute. This attribute is a tuple of strings that represent the fields on which the search will be performed:

```
search_fields = ('name', 'city')
```

Now on the admin interface, we have a search bar that we can use to search for items. Let’s type city name *Bakerview* and see what happens (see Figure 4-17).

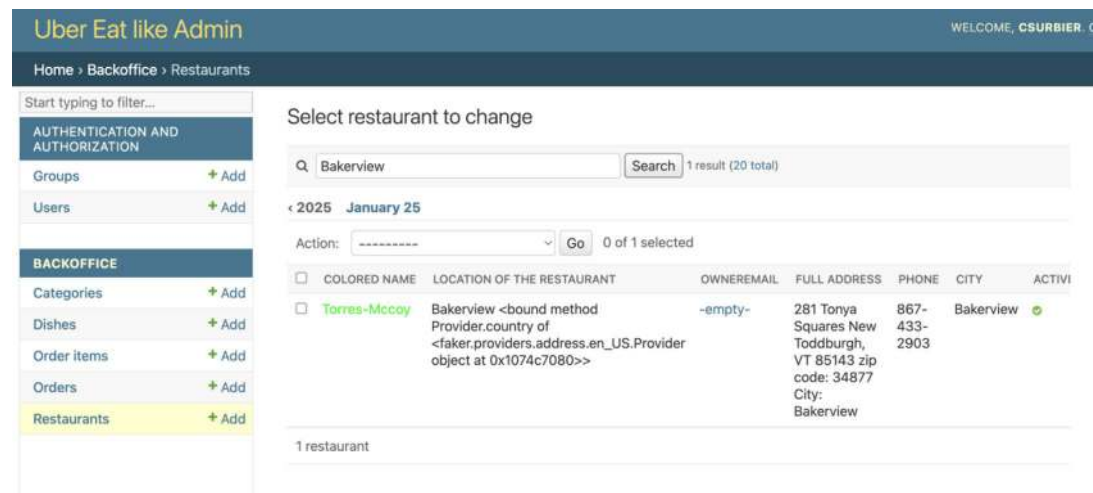


Figure 4-17. Searching the changelist using `search_fields` (example query by city)

We can also notice the number of items found in the search results: 1 result (20 total), meaning we have 1 item that matches the search query.

Let's now type the beginning of a restaurant name *Ant* and see what happens (see Figure 4-18).

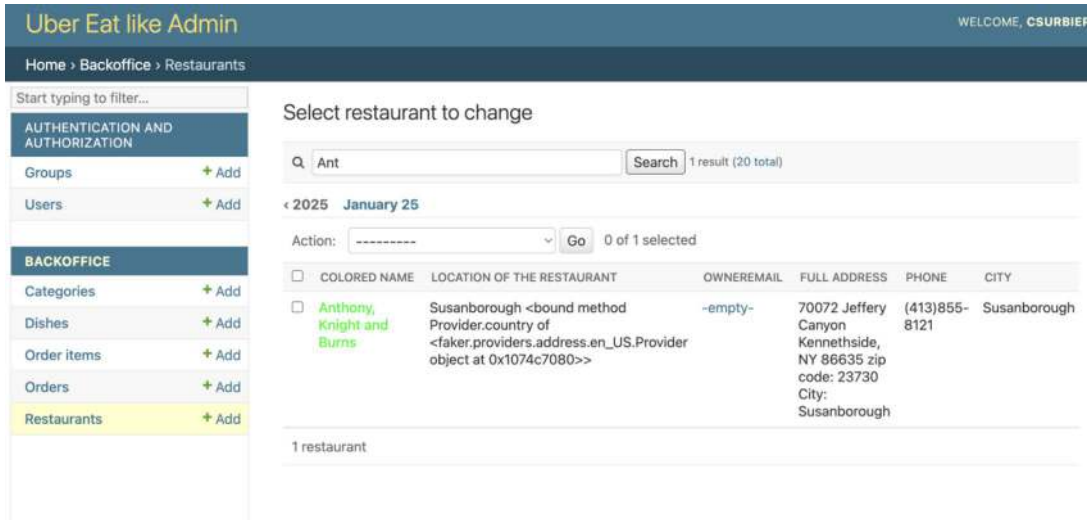


Figure 4-18. Partial text search in the changelist search bar (case-insensitive by default)

The search bar is not case sensitive, so we can search for *bakerview* or *ant* and it will still work. It's because **Django** is using the `icontains` operator to search for items. If you want to customize the search operator, you can use the `search_fields` attribute like this:

```
search_fields = ('name__exact', 'city')
```

With the `__exact` operator, we are telling **Django** to use the exact operator to search for items. Which means for *Ant*, it will search for name equals exactly to *Ant*.

Here is the list of some operators you can use:

- `__exact`
- `__iexact`
- `__istartswith`

- `__iendswith`
- `__contains`
- `__icontains`

Of course it's still possible to use the double underscore syntax to search on foreign keys; for example, if we want to search on the owner email, we can do the following:

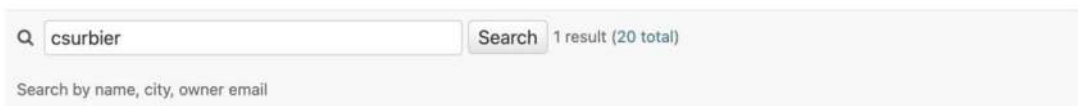
```
search_fields = ('name__exact', 'city', 'owner__email',)
```

Search Fields Help Text

To help the user to know what fields are searchable, we can use the `search_help_text` attribute. This attribute is a string that will be displayed below the search bar (see Figure 4-19):

```
search_help_text = _("Search by name, city, owner email")
```

Select restaurant to change



The screenshot shows the Django admin interface. At the top, there is a text label "Select restaurant to change". Below it is a search bar with a magnifying glass icon on the left, a text input field containing "csurbier", a "Search" button, and a status indicator "1 result (20 total)". Below the search bar, the text "Search by name, city, owner email" is displayed, which is the help text defined in the code.

Figure 4-19. *search_help_text displayed under the admin search box*

Change Form

The change form is the view that is displayed when you click on a row in the list view. It is used to edit the data of a single object.

While being able to edit an object without code is already quite magical, let's now see how we can go even further by customizing things.

fields Attribute

By default, the fields displayed on the change form are the one defined in the model instance. We can modify the fields displayed on the change form by using the `fields` attribute:


```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Full address", ordering="address")
    def full_address(self, obj):
        return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

    @admin.display(ordering="owner__email")
    def ownerEmail(self, obj):
        return obj.owner.email

    fields = ["name", "phone",]
```

With this code, we can only edit the name and phone fields, as shown in Figure 4-20.

The screenshot shows the Django Admin interface for a restaurant management system. The main content area is titled 'Change restaurant' and shows the details for 'Byrd Ltd from Holtshire'. There are two input fields: 'Name' with the value 'Byrd Ltd' and 'Phone' with the value '218.842.9134'. Below the fields are three buttons: 'SAVE', 'Save and add another', and 'Save and continue editing'. A red 'Delete' button is also visible. The left sidebar contains a list of menu items, with 'Restaurants' highlighted. The top navigation bar shows the site name 'Uber Eat like Admin' and a user profile for 'CSURBIER'.

Figure 4-20. Restricting fields shown on the change form with the `fields` attribute

readonly_fields

We can also use the `read_only_fields` attribute to make a field read-only:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    fields = ["name", "phone",]
    readonly_fields = ['phone']
```

and now **phone** is read-only (not editable), as shown in Figure 4-21.

The screenshot shows the 'Uber Eat like Admin' interface. The top navigation bar includes the title 'Uber Eat like Admin' and a user welcome message 'WELCOME, CSURBIER. CHANGE PASSWORD / LOG OUT'. Below the navigation bar, the breadcrumb trail reads 'Home > Backoffice > Restaurants > Byrd Ltd from Holtshire'. On the left, a sidebar menu lists various administrative sections: 'AUTHENTICATION AND AUTHORIZATION' (Groups, Users) and 'BACKOFFICE' (Categories, Dishes, Order items, Orders, Restaurants). The main content area is titled 'Change restaurant' and shows the details for 'Byrd Ltd from Holtshire'. It includes a 'Name' field with the value 'Byrd Ltd' and a 'Phone' field with the value '218.842.9134'. At the bottom of the form, there are four buttons: 'SAVE', 'Save and add another', 'Save and continue editing', and a red 'Delete' button. A 'HISTORY' button is also visible in the top right corner of the form area.

Figure 4-21. Making a field read-only on the change form with `readonly_fields`

exclude

We can also use the `exclude` attribute to exclude fields from the change form:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    exclude = ['phone', 'country', ]
```

The **change_form** will display all model fields except the **phone** and **country** fields, because we excluded them (Figure 4-22).

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

BACKOFFICE

Categories + Add

Dishes + Add

Order items + Add

Orders + Add

Restaurants + Add

Name: Byrd Ltd

Image: Choisir un fichier Aucun fichier choisi

Address: 06624 Michelle Coves
Lake Matthew, GU 36371

Zip code: 12775

City: Holtshire

Owner: csurbier

☐ Active

SAVE Save and add another Save and continue editing Delete

Figure 4-22. *Excluding fields from the change form with `exclude`*

To group some fields on the same line, we can use a tuple (i.e., fields between parenthesis):

```
fields = ['name', ('phone', 'country'),]
```

Phone and country will be displayed on the same line (Figure 4-23).

Uber Eat like Admin

Home > Backoffice > Restaurants > Byrd Ltd from Holtshire

WELCOME, CSURBIER CHANGE PASSWORD / LOG OUT

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

BACKOFFICE

Categories + Add

Dishes + Add

Order items + Add

Orders + Add

Restaurants + Add

Change restaurant

Byrd Ltd from Holtshire

Name: Byrd Ltd

Phone: 218.842.9134

Country: <bound method Provider.country of <faker>

SAVE Save and add another Save and continue editing Delete

Figure 4-23. *Grouping fields on one line using tuples in the fields definition*

fieldsets

For more complex layouts, we can use the `fieldsets` attribute. It is a list of tuples; each tuple (a section in our `change_form`) contains two elements:

- The first element is the title of the fieldset.
- The second element is a list of fields to display in the fieldset.

Let's have a look at the following example (see Figure 4-24 for result):

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {
            'fields': ('name', 'image', 'address', 'zip_code', 'city',
                       'country', 'phone', )
        }),
        ('Advanced options', {
            'classes': ('collapse',),
            'fields': ('is_active', 'owner',),
        }),
    )
```

Uber Eat like Admin

WELCOME, CSURBIER. CHANGE PASSWORD / LOG OUT

Home > Backoffice > Restaurants > Byrd Ltd from Holtshire

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups Add

Users Add

BACKOFFICE

Categories Add

Dishes Add

Order items Add

Orders Add

Restaurants Add

Change restaurant

Byrd Ltd from Holtshire

HISTORY

Name: Byrd Ltd

Image: Choisir un fichier Aucun fichier choisi

Address: 06624 Michelle Coves
Lake Matthew, GU 36371

Zip code: 12775

City: Holtshire

Country: <bound method Provider.country of <faker>

Phone: 218.842.9134

Advanced options

SAVE

Save and add another

Save and continue editing

Delete

Figure 4-24. Change form layout using fieldsets (sectioned form)

Byrd Ltd from Holtshire

My first section

This is the first section of the form.

Name:

Image: Aucun fichier choisi

Address:

06624 Michelle Coves
Lake Matthew, GU 36371

Zip code:

City:

Country:

Phone:

Figure 4-25. Collapsible “Advanced options” fieldset using the `collapse` class

We are displaying the fields: name, image, address, zip_code, city, country, and phone in the first section and then we have the `is_active` and `owner` fields in the second section called Advanced options with a collapse button because of the `classes` attribute.

If we want to give a name to our first section (see Figure 4-26) such as My first section, we can write

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    fieldsets = (
        ("My first section", {
            'fields': ('name', 'image', 'address', 'zip_code', 'city',
                       'country', 'phone', )
        }),
        ('Advanced options', {
            'classes': ('collapse',),

```

```

    'fields': ('is_active','owner'),
  }},
)

```

The screenshot shows a web application interface for managing restaurants. The main content area is titled 'Change restaurant' and 'Byrd Ltd from Holtshire'. A custom fieldset titled 'My first section' contains several form fields: 'Name' (Byrd Ltd), 'Image' (Choir un fichier), 'Address' (06624 Michelle Coves, Lake Matthew, GU 36371), 'Zip code' (12775), 'City' (Holtshire), 'Country' (a placeholder text), and 'Phone' (218.842.9134). Below this is an 'Advanced options' section. At the bottom of the form are four buttons: 'SAVE', 'Save and add another', 'Save and continue editing', and 'Delete'.

Figure 4-26. Custom fieldset title (“My first section”) in the change form

We can also use the description attribute to add a description to a fieldset section (as shown on Figure 4-27):

```

fieldsets = (
    ("My first section", {
        'fields': ('name', 'image', 'address', 'zip_code', 'city',
        'country', 'phone', ),
        'description': 'This is the first section of the form.'
    }),
    ('Advanced options', {
        'classes': ('collapse',),
        'fields': ('is_active','owner',),
    }),
)

```

Byrd Ltd from Holtshire

My first section

This is the first section of the form.

Name:

Byrd Ltd

Image:

Choisir un fichier

Aucun fichier choisi

Address:

06624 Michelle Coves
Lake Matthew, GU 36371

Zip code:

12775

City:

Holtshire

Country:

<bound method Provider.country of <faker.␣

Phone:

218.842.9134

Figure 4-27. Fieldset description text displayed in the change form

Like with **fields**, we can group some fields on the same line using parentheses (see Figure 4-28):

```
fieldsets = (  
    ("My first section", {  
        'fields': ('name', 'image', ('address', 'zip_code', 'city'),  
        'country', 'phone', ),  
        'description': 'This is the first section of the form.'  
    }},  
    ('Advanced options', {  
        'classes': ('collapse',),  
        'fields': ('is_active', 'owner',),  
    }},  
)
```


Change restaurant

Byrd Ltd from Holtshire

HISTORY

My first section

This is the first section of the form.

Name:	Byrd Ltd		
Image:	Choisir un fichier	Aucun fichier choisi	
Address:	06624 Michelle Coves
Lake Matthew, GU 36371		
Zip code:	12775	City:	Holtshire
Country:	bound method Provider.country of <faker.		
Phone:	218.842.9134		

► Advanced options

SAVE

Save and add another

Save and continue editing

Delete

Figure 4-28. Grouping multiple fields on a single row within fieldsets

classes

The `classes` attribute is used to add classes to the fieldset. For example, we added the `collapse` class to the Advanced options fieldset to make it collapsible.

By default, Django Admin provides two useful built-in CSS classes: `collapse` and `wide`.

It's possible to add our own classes to the admin site CSS stylesheet and we will study how to do that in the next chapter.

CHAPTER 5

Actions and Operations

Django Admin’s action system allows us to perform operations on multiple objects at once. This powerful feature helps streamline administrative tasks and batch processing.

Built-In Actions

By default, Django Admin comes with a “Delete selected objects” action. This action appears as a dropdown menu above the list view when one or more objects are selected (Figure 5-1).

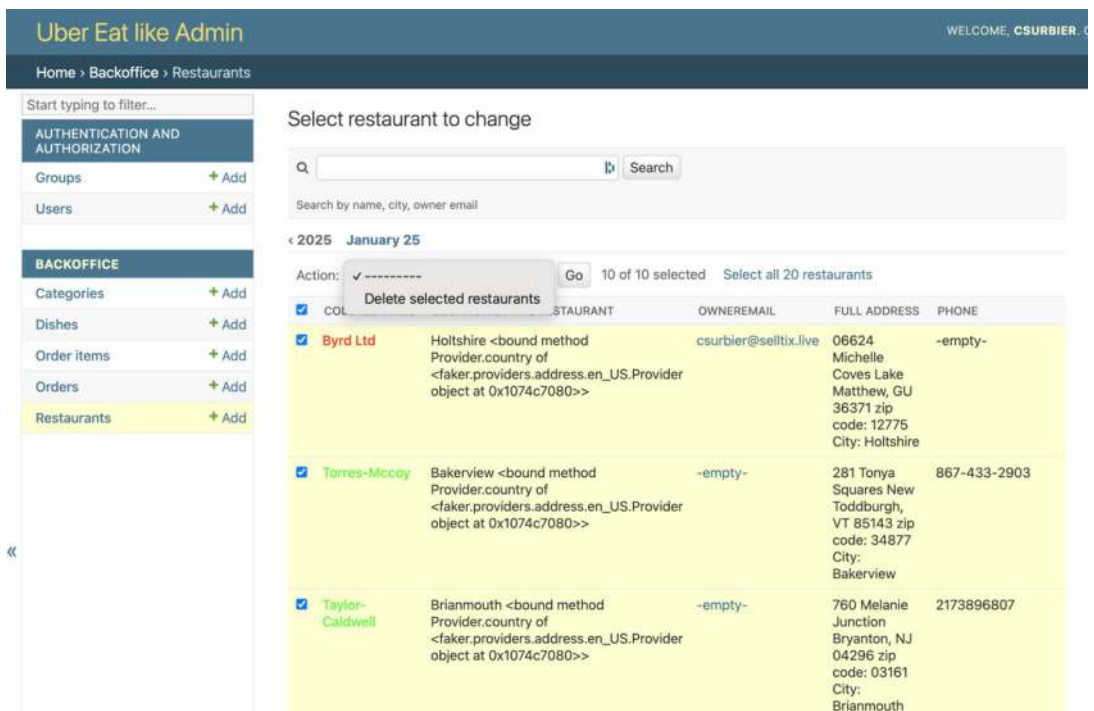


Figure 5-1. Built-in “Delete selected objects” action dropdown on the changelist

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    actions = ['delete_selected'] # This is the default and we don't need
    to specify it
```

We can disable this delete action by setting the actions attribute to an empty list (see Figure 5-2):

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    actions = None
```

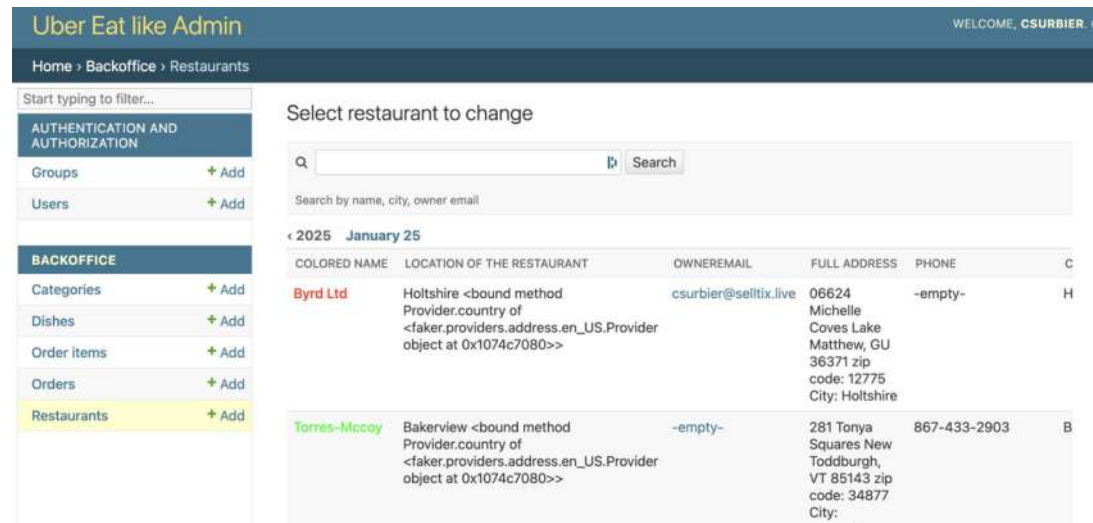


Figure 5-2. Actions dropdown removed after disabling actions on the *ModelAdmin*

Custom Actions

We can create our own actions by using the `@admin.action` decorator. This decorator takes a description as a parameter, which will be displayed in the dropdown menu. Then we need to define a function that will be called when the action is triggered. And finally, to register the action, we need to add it to the actions attribute of the *ModelAdmin* class.

Let's create a custom action `make_inactive` as example that will set the `is_active` field to `False` for all the selected objects:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    actions = ['make_inactive']

    @admin.action(
        description="Mark selected restaurants as inactive"
    )
    def make_inactive(self, request, queryset):
        updated = queryset.update(is_active=False)
        self.message_user(
            request,
            f"{updated} restaurants were successfully marked as inactive."
        )
```

The action method receives

- `self`: The current admin instance
- `request`: The current request
- `queryset`: The queryset of the selected objects

And now we can see and use this action (Figure 5-3).

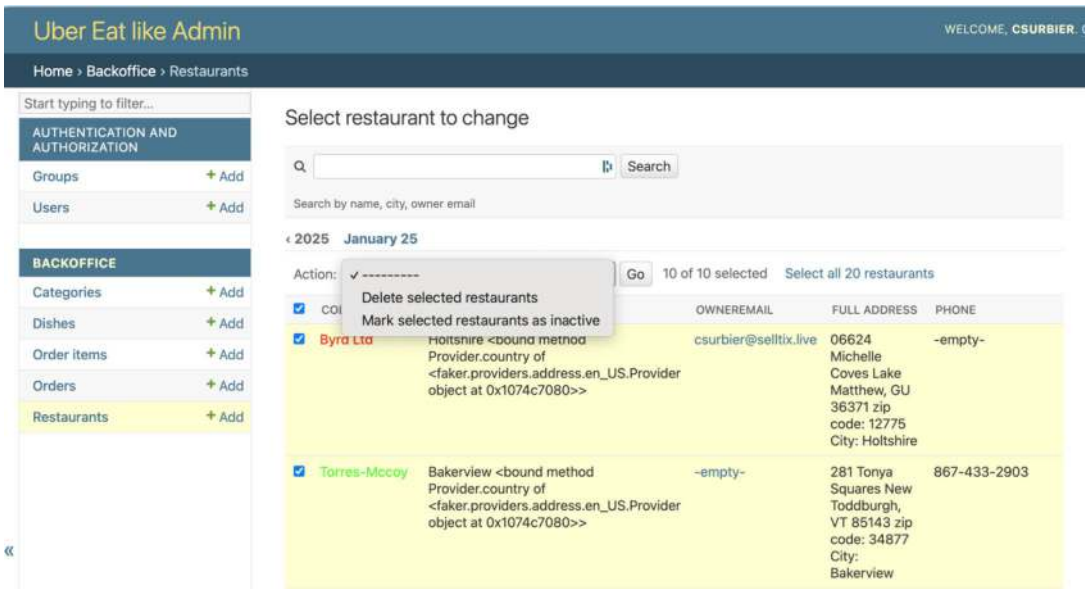


Figure 5-3. Custom action added to the actions dropdown (batch update example)

This is really easy to create custom actions. We will learn in the next chapter how to create more advanced actions.

Action Responses and Messages

We can use the `message_user` method to display a message to the user after an action is performed. This method takes the parameters

- `request`: The current request
- `message`: The message to display
- `level`: The level of the message (default is SUCCESS)
 - SUCCESS
 - INFO
 - WARNING
 - ERROR

```

self.message_user(
    request,
    f'{updated} restaurants were successfully marked as inactive.'
)

```

But we can also customize the response of the action. For example, if we want to download a file, we can use the response method. Let's first import the **csv** library:

```

import csv
from django.http import HttpResponse

```

and then write an action that will export the selected restaurants to a CSV file:

```

@admin.action(description="Export selected restaurants to CSV")
def export_to_csv(self, request, queryset):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment;
filename="restaurants.csv"'

    writer = csv.writer(response)
    writer.writerow(['Name', 'Address', 'Phone', 'Owner'])

    for restaurant in queryset:
        writer.writerow([
            restaurant.name,
            restaurant.address,
            restaurant.phone,
            restaurant.owner.email
        ])

    return response

```

We specify that the response is a CSV file with the `content_type` parameter. Then we specify the filename with the Content-Disposition header. Finally, we create a writer object and write the header row. Then we loop over the queryset and write the data to the CSV file. Finally, we return the response object.

If we select this action, we will see our file being downloaded in the browser (as shown on Figure 5-4).

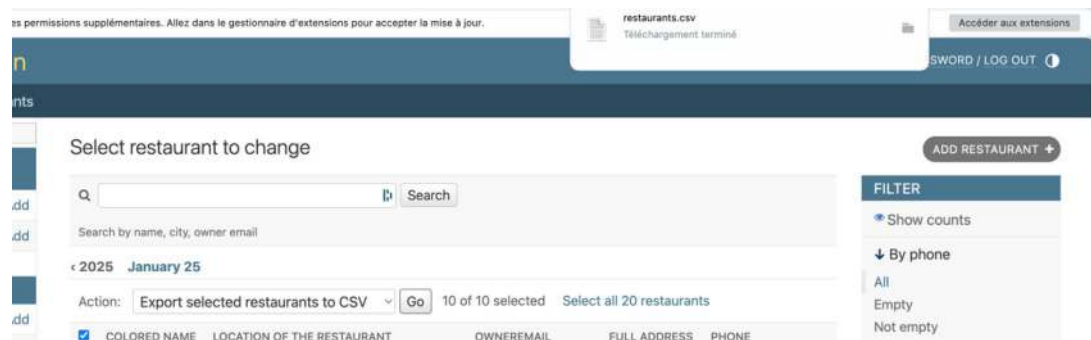


Figure 5-4. Custom action returning a file download response (CSV export)

We could also create custom response pages for our actions. We will learn how to do this in the next chapter.

CHAPTER 6

Forms and Validation

Django Admin provides extensive form customization capabilities that allow us to control how data is entered and validated on `change_form`.

Form Customization

To customize the form used in the admin interface, we create a `ModelForm` class:

```
from django import forms

class RestaurantForm(forms.ModelForm):
    confirm_phone = forms.CharField(max_length=20)

    class Meta:
        model = Restaurant
        fields = '__all__'

    def clean(self):
        cleaned_data = super().clean()
        phone = cleaned_data.get('phone')
        confirm_phone = cleaned_data.get('confirm_phone')

        if phone != confirm_phone:
            raise ValidationError("Phone numbers must match")

        return cleaned_data
```

Note If a `ModelForm` is only used in the Django Admin, we can usually omit `Meta.model`, because the `ModelAdmin` already provides the model; keep `Meta.model` only if the form is reused outside the admin.

and then link it to our ModelAdmin:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    form = RestaurantForm
```

The form above will display the form with a new `confirm_phone` field. If the phone numbers don't match, we will get a validation error. The validation occurs in the `clean` method of the form. The `cleaned_data` dictionary contains the cleaned data of the form. A cleaned data is a data that has been validated and is ready to be used. To reject the form, we can raise a `ValidationError` with a message which will be displayed to the user on the interface.

Before going further and test the results, we need to add the new field `confirm_phone` to the list of fieldsets of our ModelAdmin:

```
fieldsets = (
    ("My first section", {
        'fields': ('name', 'image', ('address', 'zip_code', 'city'),
        'country', 'phone', 'confirm_phone', ),
        'description': 'This is the first section of the form.'
    }),
    ('Advanced options', {
        'classes': ('collapse',),
        'fields': ('is_active', 'owner',),
    }),
)
```

Now we can reload our admin interface and see the new field the error message that we raised from `ValidationError` if we entered values that don't match (Figure 6-1).

The screenshot shows a web application titled 'Uber Eat like Admin'. The top navigation bar includes 'Home > Backoffice > Restaurants > Byrd Ltd from Holtshire'. The left sidebar has sections for 'AUTHENTICATION AND AUTHORIZATION' (Groups, Users) and 'BACKOFFICE' (Categories, Dishes, Order items, Orders, Restaurants). The main content area is titled 'Change restaurant' and shows the details for 'Byrd Ltd from Holtshire'. A red error message 'Please correct the error below.' is displayed. Below it, a message says 'Phone numbers must match'. The form fields are: Name (Byrd Ltd), Image (Choisir un fichier), Address (06624 Michelle Coves, Lake Matthew, GU 36371), Zip code (12775), City (Holtshire), Country (<bound method Provider.country of <faker>), Phone (test), and Confirm phone (attr). At the bottom, there are buttons for 'SAVE', 'Save and add another', 'Save and continue editing', and 'Delete'.

Figure 6-1. Custom `ModelForm` in admin with an extra confirmation field and validation error display

This could be really useful to validate data before saving it to the database.

Validation Rules

Previously, we saw how to check fields using the `clean` method, but we can also add validation rules to our form directly to a specific field.

We can do this by adding a method to our form class which will be named by convention `clean_`. For example, if we want to validate the phone number, we can add the following method to our form class:

```
def clean_phone(self):
    phone = self.cleaned_data.get('phone')
    confirm_phone = self.cleaned_data.get('confirm_phone')
    if not phone.startswith('+237'):
```

```

raise forms.ValidationError("Phone number must start with +237")

if phone != confirm_phone:
    raise forms.ValidationError("Phone numbers must match")
return phone

```

and remove or comment our previous `clean` method because we are doing the same thing in the `clean_phone` method. With these new checks, we can see that we have our validation error (Figure 6-2).

Change restaurant

Byrd Ltd from Holtshire HISTORY

Please correct the error below.

My first section

This is the first section of the form.

Name:

Image: Aucun fichier choisi

Address:

Zip code: City:

Country:

Phone: Phone number must start with +237

Confirm phone:

Advanced options

Figure 6-2. Field-level validation with `clean_<fieldname>` showing an error on the form

CHAPTER 7

Inline Editing

With models having a foreign key relationship, we can use the `TabularInline` or `StackedInline` classes to display the related objects in the `change_form`. In other words, since we have a foreign key relationship between the `Restaurant` and the `Category` models, we can display the categories in the `change_form` of the `Restaurant` model. This is a really useful feature.

Let's see how to do this:

```
class CategoryInline(admin.TabularInline):  
    model = Category  
    extra = 4  
  
@admin.register(Restaurant)  
class RestaurantAdmin(admin.ModelAdmin):  
    inlines = [CategoryInline]
```

which will display the categories in a tabular format, as shown in Figure 7-1.

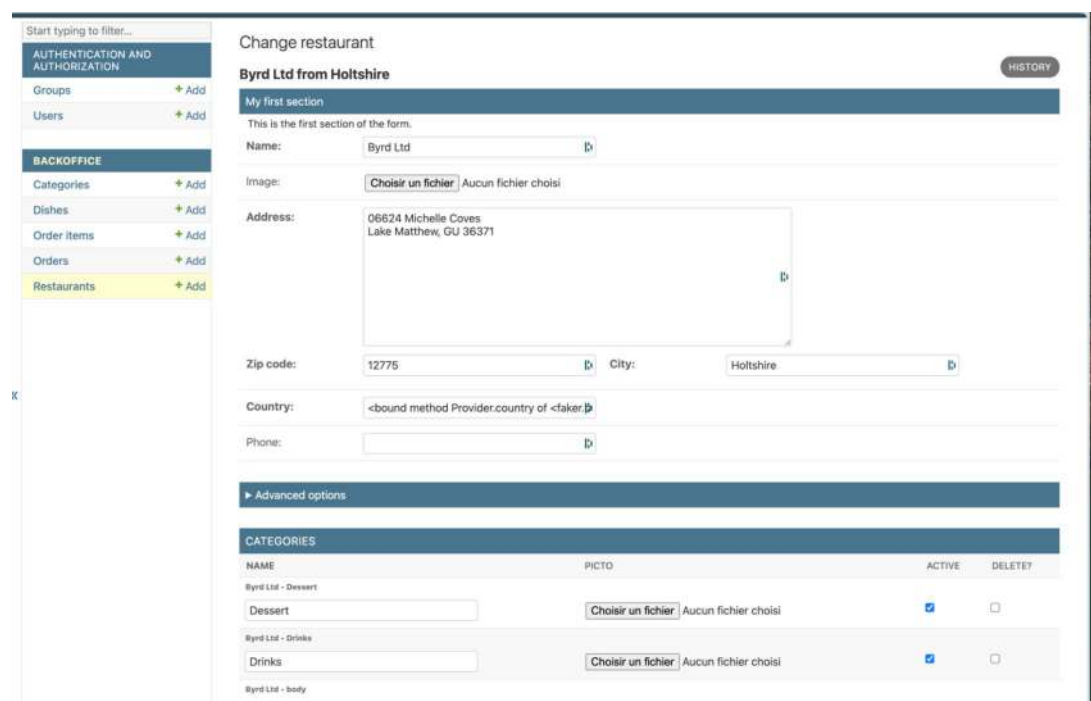


Figure 7-1. *Editing related objects inline using TabularInline*

while the StackedInline class

```
class CategoryInline(admin.StackedInline):
    model = Category
    extra = 4
```

will display the categories in a stacked format (Figure 7-2).

Figure 7-2. *Editing related objects inline using StackedInline*

We can also notice that at the end of the categories list, we have four blank lines (see Figure 7-3) which can be edited directly from our Restaurant change_form to add more categories!

Figure 7-3. *Extra blank inline forms controlled by the extra setting*

This is the purpose of the extra attribute that we can use to define the number of blank lines to display at the end of the list:

extra = 4

There is also at the end of the list a plus sign with a label “Add another category”, which on click will add another blank line. Amazing!

More basic options can be specified in the `InlineModelAdmin` class:

- `min_num`: The minimum number of objects that must be present in the inline model.
- `max_num`: The maximum number of objects that can be present in the inline model.
- `can_delete`: Whether the objects in the inline model can be deleted.
- `show_change_link`: Whether the objects in the inline model can be edited. When set to `True`, Django admin shows a “Change” link for each inline row, which opens that related object’s own change form page (instead of editing only within the inline).

We will see more advanced options in the next chapter.

Many to Many Relationships

There is one kind of relationships we didn’t talk about yet, and that is the **many to many relationship**. This is a relationship where a model can be related to many other models and vice versa. For example, a dish can have many allergens and an allergen can be related to many dishes. This is a many to many relationship.

Let’s modify our `models.py` file to add the `Allergen` model:

```
class Allergen(models.Model):
    name = models.CharField(_('name'), max_length=100) # e.g., Peanuts,
    Gluten, Lactose
    description = models.TextField(_('description'), blank=True)
    icon = models.ImageField(upload_to='allergens/', blank=True)

class Meta:
    verbose_name = _('allergen')
    verbose_name_plural = _('allergens')
```

```

def __str__(self):
    return self.name

class Dish(models.Model):
    name = models.CharField(_('name'), max_length=200)
    description = models.TextField(_('description'))
    price = models.DecimalField(_('price'), max_digits=6, decimal_places=2)
    category = models.ForeignKey(Category, on_delete=models.CASCADE,
    related_name='dishes')
    allergens = models.ManyToManyField(Allergen, blank=True, related_
    name='dishes')
    is_available = models.BooleanField(_('available'), default=True)
    image = models.ImageField(_('image'), upload_to='dishes/', blank=True)

    class Meta:
        verbose_name = _('dish')
        verbose_name_plural = _('dishes')
        ordering = ['name']

    def __str__(self):
        return f"{self.category.restaurant.name} - {self.name}"

```

and now we need to run once again our migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Looking at the admin interface (see Figure 7-4), we can see that we have a new field called allergens in the Dish model.

The screenshot shows the 'Uber Eat like Admin' interface. The top navigation bar includes the title 'Uber Eat like Admin' and a user welcome message 'WELCOME, CSURBIER' with links for 'CHANGE PASSWORD' and 'LOG OUT'. The breadcrumb trail is 'Home > Backoffice > Dishes > Add dish'. The left sidebar contains a search bar and two main sections: 'AUTHENTICATION AND AUTHORIZATION' with links for 'Groups' and 'Users', and 'BACKOFFICE' with links for 'Categories', 'Dishes' (highlighted), 'Order Items', 'Orders', and 'Restaurants'. The main content area is titled 'Add dish' and contains the following form fields: 'Name' (text input), 'Description' (text area), 'Price' (text input), 'Category' (dropdown menu), 'Allergens' (multi-select field with a red box around it), and 'Image' (file upload button). At the bottom of the form are three buttons: 'SAVE', 'Save and add another', and 'Save and continue editing'.

Figure 7-4. Many-to-many field displayed on the Dish form (allergens multi-select)

First, let's add the Allergen model to our admin interface:

```
@admin.register(Allergen)
class AllergenAdmin(admin.ModelAdmin):
    pass
```

We can notice that we didn't define any attribute in the AllergenAdmin class (by using the **pass** instruction). This is because we want to use the default behavior of the ModelAdmin class.

Let's create a Peanuts allergen (Figure 7-5).

The screenshot shows the 'Add allergen' form in the 'Uber Eat like Admin' interface. The form is titled 'Add allergen' and is located in the 'Backoffice > Allergens > Add allergen' section. The left sidebar shows the navigation menu with 'Allergens' highlighted under the 'BACKOFFICE' section. The form fields are as follows:

- Name:** A text input field containing 'Peanuts'.
- Description:** A large text area containing 'Peanuts'.
- Icon:** A file upload field with the label 'Choir un fichier' and a button 'Aucun fichier choisi'.

At the bottom of the form, there are three buttons: 'SAVE', 'Save and add another', and 'Save and continue editing'.

Figure 7-5. Creating an Allergen object in the admin (example: Peanuts)

And now we can add it to our Dish model (Figure 7-6).

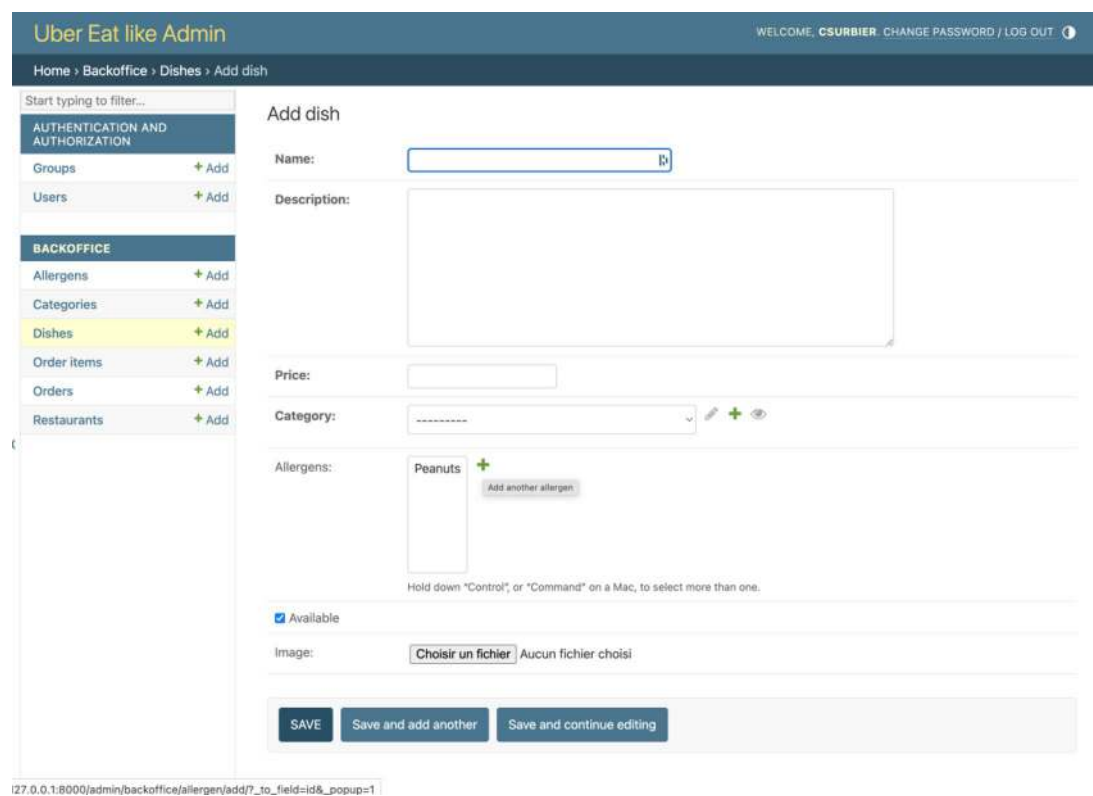


Figure 7-6. *Selecting related allergens on the Dish change form (many-to-many selection UI)*

We can notice that we have a new field called allergens in the Dish model with our existing Peanuts value. This field is a many to many relationship field. We also have a + button which means that we can add multiple allergens to a dish.

The default admin behavior is to use a multiple-select boxes for this kind of fields.

For selecting values for a **many to many field**, we have two options in the admin: filter_horizontal and filter_vertical attributes (please refer to Figure 7-7 or Figure 7-8 to check the rendering).

```
@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'price', 'is_available',)
```

```
list_filter = ('is_available', 'category')
search_fields = ('name', 'description')
filter_horizontal = ('allergens',)
```

The screenshot shows the 'Add dish' form in a Django Admin interface. The left sidebar contains a navigation menu with sections: 'AUTHENTICATION AND AUTHORIZATION' (Groups, Users) and 'BACKOFFICE' (Allergens, Categories, Dishes, Order items, Orders, Restaurants). The 'Dishes' section is highlighted. The main form area is titled 'Add dish' and includes fields for Name, Description, Price, and Category. The 'Allergens' field is a many-to-many relationship, implemented using the `filter_horizontal` widget. It consists of two side-by-side panels: 'Available allergens' and 'Chosen allergens'. Both panels have a search filter and a list of allergens (currently showing 'Peanuts'). Below the panels are buttons for 'Choose all' and 'Remove all'. A note below the buttons states: 'Hold down "Control", or "Command" on a Mac, to select more than one.' At the bottom of the form, there is a checkbox for 'Available' and an 'Image' field.

Figure 7-7. Improved many-to-many selection using `filter_horizontal`

or

```
@admin.register(Dish)
```

```
class DishAdmin(admin.ModelAdmin):
```

```
    list_display = ('name', 'category', 'price', 'is_available',)
    list_filter = ('is_available', 'category')
    search_fields = ('name', 'description')
    filter_vertical = ('allergens',)
```

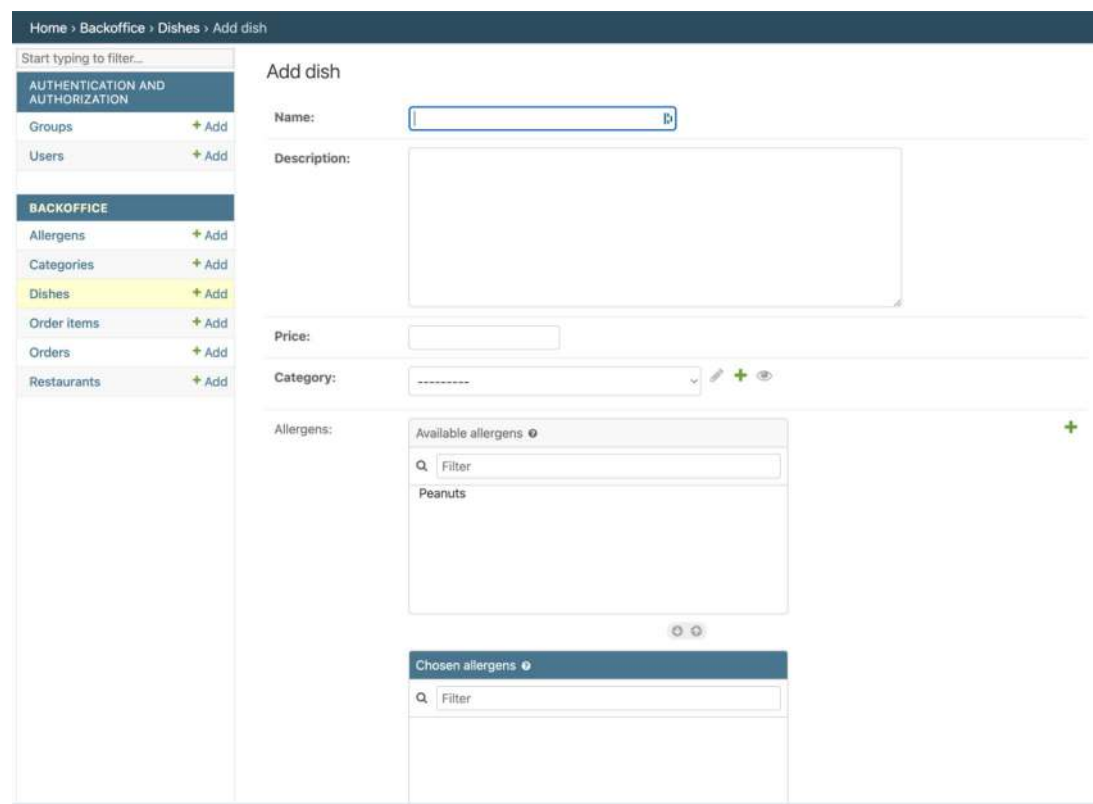


Figure 7-8. Improved many-to-many selection using `filter_vertical`

These `filter_horizontal` or `filter_vertical` attributes are creating an intuitive interface for managing many-to-many relationships with two boxes side by side (horizontal) or stacked (vertical), allowing easy selection/deselection of related objects. We also have a search box included, which is useful to filter when there is lot of data in the model. So these options only change the admin form widget/UI for the many to many field, but don't change the underlying many-to-many relationship logic or how it's saved.

CHAPTER 8

Performance Optimization

Django Admin's performance can be significantly improved through various optimization techniques and should be done before going to production!

Let's explore the key strategies.

Query Optimization

Using `select_related` and `prefetch_related` can dramatically reduce the number of database queries and must be used **if** the admin page accesses related objects. It will reduce the **N+1 queries problem**: the N+1 query problem happens when Django first runs one query to fetch a list of objects, then runs an extra query for each object to fetch related data, resulting in 1+N total queries.

`select_related`

`select_related` is used to reduce the number of queries by joining related tables in a single query. It's particularly useful for one-to-one and one-to-many relationships.

As reminder, we added in our `RestaurantAdmin` and the `list_display` attribute, the possibility to show the email of the owner of the restaurant:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    list_display = ('colored_name', 'location', 'ownerEmail', 'full_address',
                   'phone', 'city', 'is_active', 'created_at')
```

Without using the `select_related` method, we will have lot of database queries. In fact, Django will make one query to fetch all restaurants and then N queries (one per restaurant) to fetch each owner's email. In production environment, this is **not acceptable**.

To better understand this, let's use the Django toolbar, which is an invaluable tool for debugging and optimizing Django applications. It provides detailed insights into the performance of your application, including the number of database queries, their execution time, and more.

To install the Django toolbar, let's add it to our requirements.txt file

```
django-debug-toolbar==5.0.1
```

and then install it:

```
pip install -r requirements.txt
```

Then, we need to add some configuration instructions to our settings.py file. To know more about the Django toolbar, you can visit the official documentation: [Django Debug Toolbar](https://django-debug-toolbar.readthedocs.io/en/latest/) (<https://django-debug-toolbar.readthedocs.io/en/latest/>):

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'backoffice',  
    'debug_toolbar',  
]  
  
MIDDLEWARE = [  
    'django.middleware.locale.LocaleMiddleware',  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
    'debug_toolbar.middleware.DebugToolbarMiddleware',  
]  
]
```

```
INTERNAL_IPS = [
    # ...
    "127.0.0.1",
    # ...
]
```

INTERNAL_IPS lists the IP addresses Django treats as internal/trusted; the Django Debug Toolbar uses it to decide whether to show its UI for a given request.

and to our `urls.py` file:

```
"""
URL configuration for djangoadmin project.
"""

from django.conf import settings
from django.contrib import admin
from django.urls import include, path

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

if settings.DEBUG:
    urlpatterns += [path('__debug__/', include('debug_toolbar.urls'))]
```

Now, if we reload your admin interface, we should see the toolbar on the right side (Figure 8-1).

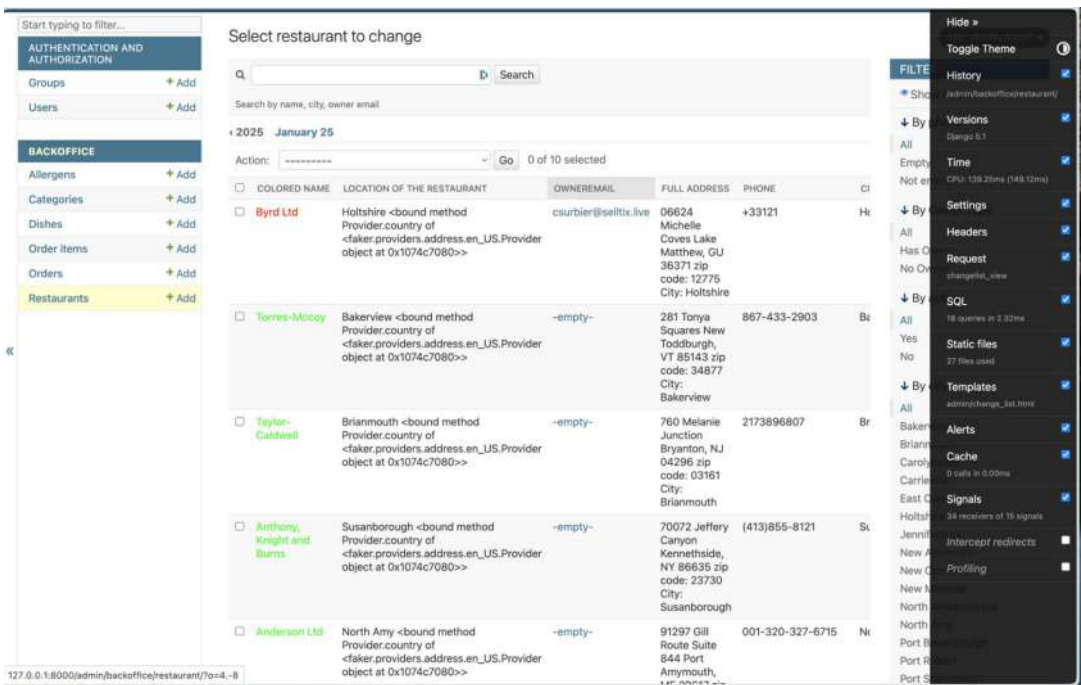


Figure 8-1. Django Debug Toolbar visible in the admin for profiling queries

Let's go to the Restaurant menu item and now open the **SQL** section of the toolbar (Figure 8-2).



Figure 8-2. Debug Toolbar SQL panel showing duplicate queries (baseline before optimization)

As we can see, a lot of (duplicate) queries are made to fetch the restaurant's owner email (from the **auth_user** model).

To fix this, we can use the `list_select_related` attribute and specify the related model (**owner**):

```
@admin.register(Restaurant)
```

```
class RestaurantAdmin(admin.ModelAdmin):
```

```
    list_display = ('colored_name', 'location', 'ownerEmail', 'full_address',
                    'phone', 'city', 'is_active', 'created_at')
```

```
    list_select_related = ('owner',)
```

Now, if we reload the page, as shown on Figure 8-3 we should see that few queries have been made and only **one query** to fetch the restaurant's owner email using the SQL instruction **inner_join**. It's a single JOIN query fetching restaurants WITH their owners.



Figure 8-3. Reduced query count after enabling `list_select_related` for the owner relationship

Using the **Django toolbar** can be very useful to understand what is happening under the hood and to optimize our queries. It should be always used to optimize queries before going on production.

Let's have a look at our DishAdmin class:

```
@admin.register(Dish)
```

```
class DishAdmin(admin.ModelAdmin):
```

```
    list_display = ('name', 'category', 'price', 'is_available',)
```

```
    list_filter = ('is_available', 'category')
```

```
    search_fields = ('name', 'description')
```

```
    filter_horizontal = ('allergens',)
```

And let's go to the Dish menu item to check the number of queries made (Figure 8-4).

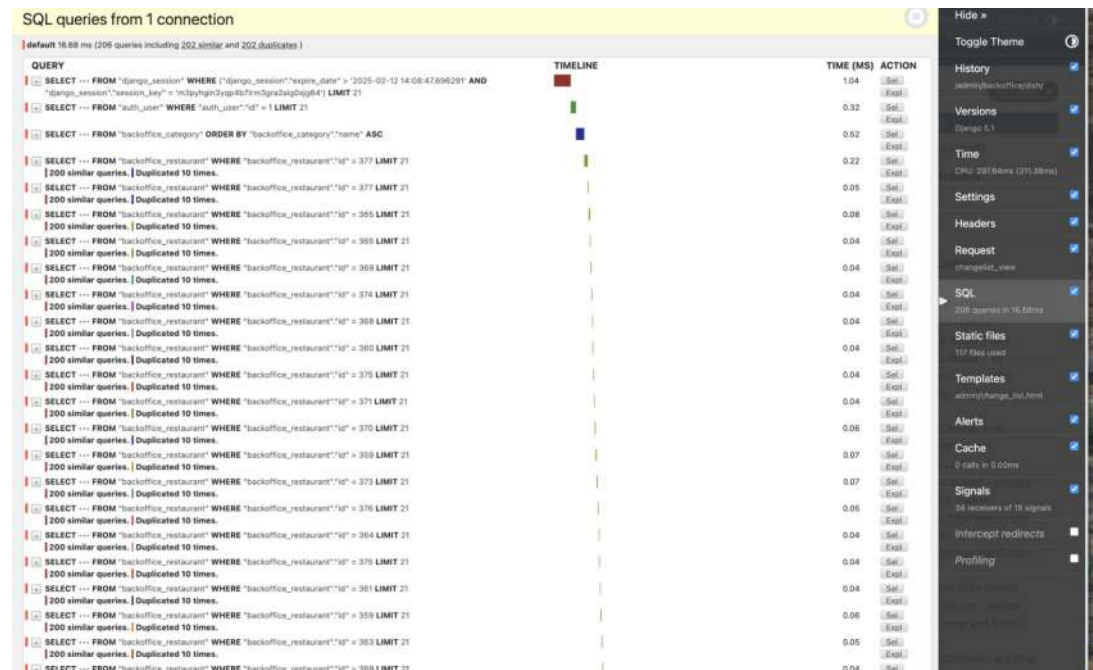


Figure 8-4. Dish changelist query count before optimization (many queries)

Wow, so many queries! As Category is a foreign key in the Dish model, we can add the `list_select_related` attribute to reduce the number of queries (see Figure 8-5):

```
admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'price', 'is_available',)
    list_filter = ('is_available','category',)
    list_select_related = ('category',)
    search_fields = ('name', 'description')
    filter_horizontal = ('allergens',)
```

If we reload the page, we could see that the number of queries has not been reduced; it's even the opposite!

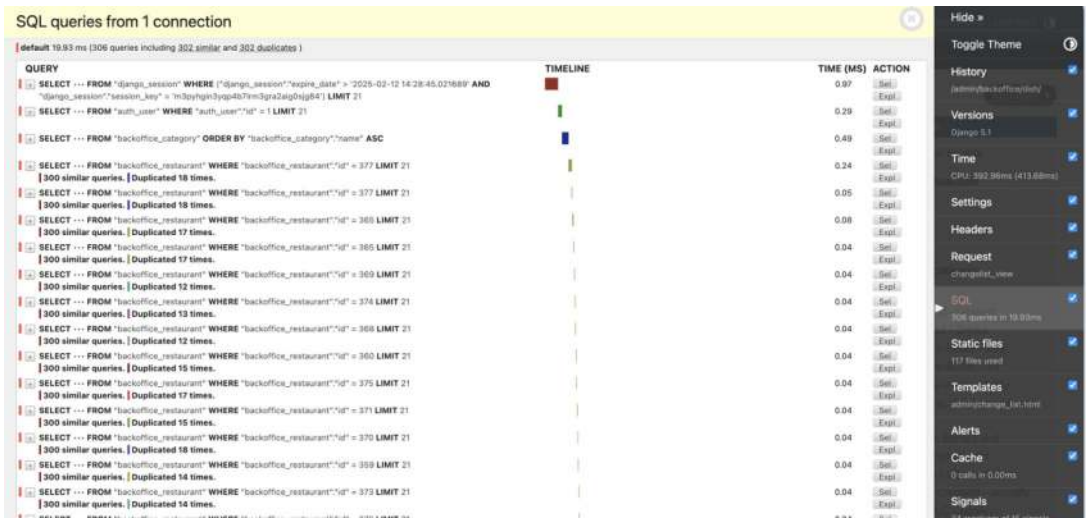


Figure 8-5. Query count impact after adding `list_select_related` without optimizing filter lookups

Remember the `list_filter` attribute? We used it to filter the dishes by category. The `list_filter` operates independently of `list_select_related`. When Django builds the filter choices for the category field, it makes separate queries to get all possible values, regardless of the `list_select_related` setting! This is quite an issue for performance. To fix this, we need to create a custom filter as we learned and override the lookups method, to get the categories with the restaurants using the `select_related` attribute.

```
class CategoryFilter(admin.SimpleListFilter):
    title = 'category'
    parameter_name = 'category'

    def lookups(self, request, model_admin):
        categories = Category.objects.select_related('restaurant').all()
        return [(c.id, str(c)) for c in categories]

    def queryset(self, request, queryset):
        if self.value():
            return queryset.filter(category_id=self.value())
        return queryset
```

```
@admin.register(Allergen)
class AllergenAdmin(admin.ModelAdmin):
    pass

@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'price', 'is_available',)
    list_filter = ('is_available', CategoryFilter)
    list_select_related = ('category',)
    search_fields = ('name', 'description')
```

OK now, if we reload, the number of queries has dropped significantly (from 306 to 106 with my sample data) (Figure 8-6).

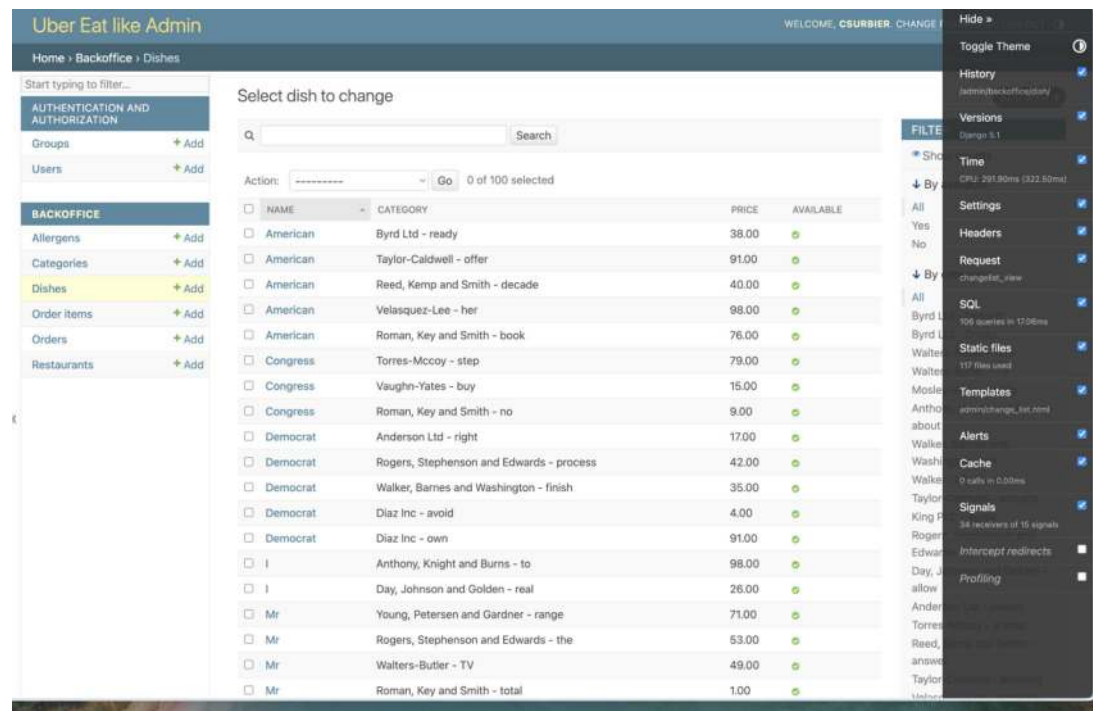


Figure 8-6. Query count reduced after optimizing filter lookups and related fetching

But we have still lot of queries to get the restaurant’s data! Remember that our Category model has a foreign key to the Restaurant model? Let’s try the double underscore notation (__) to also fetch the restaurant’s data in one query:

```
@admin.register(Dish)
```

```
class DishAdmin(admin.ModelAdmin):
```

```
    list_display = ('name', 'category', 'price', 'is_available',)
```

```
    list_filter = ('is_available', CategoryFilter)
```

```
    list_select_related = ('category', 'category__restaurant',)
```

```
    search_fields = ('name', 'description')
```

```
    filter_horizontal = ('allergens',)
```

And voilà, see Figures 8-7 and 8-8.

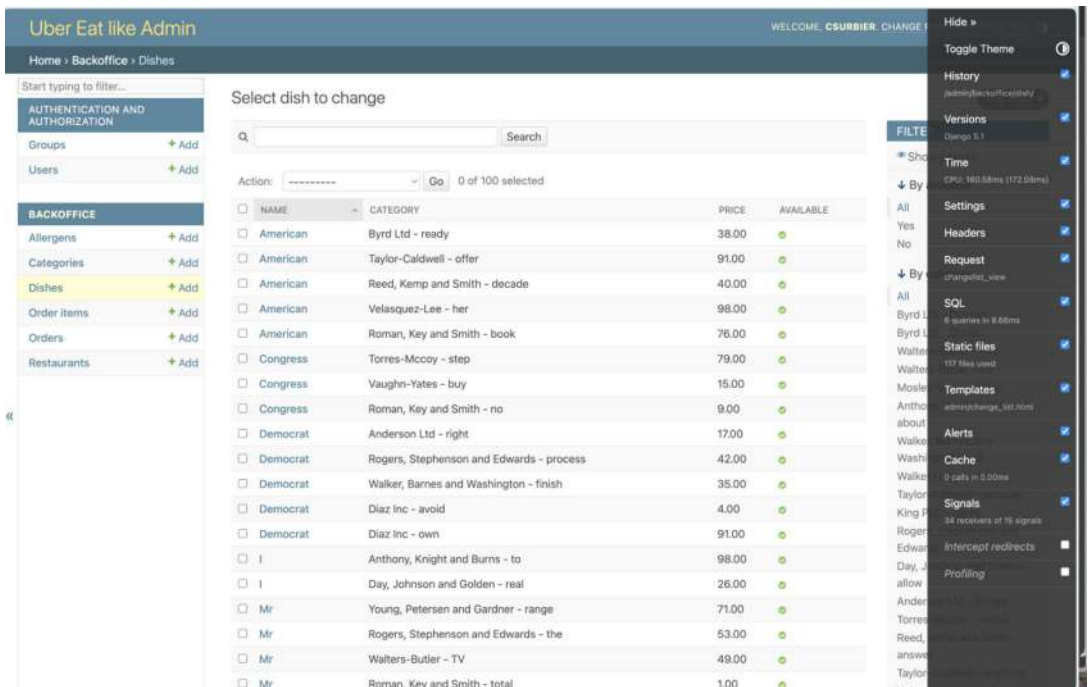


Figure 8-7. Further query reduction by selecting nested relationships (*category__restaurant*)



Figure 8-8. Debug Toolbar SQL details after adding nested list_select_related joins

Using just `list_select_related = ('category',)` will only fetch the immediate Category data in a single JOIN query. Whereas using `list_select_related = ('category', 'category__restaurant',)` fetches both the Category and its related Restaurant data in a single query with multiple JOINS.

This can be hard to not forget relationships when using `list_select_related`, and it's not always easy to know which relationships to use. That is why using the **Django toolbar** is such a great tool and a must use before going to production. Saving tons of queries will **reduce the load on the database** and improve the performance of your application.

prefetch_related

`prefetch_related` is used to reduce the number of queries by fetching related objects in a single query, as we have seen with `select_related`. However, there is no shortcut in the admin for this method, so we need to implement it through the `get_queryset` method, which we will override in our `DishAdmin` class like this:

```
@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'price', 'is_available',)
    list_filter = ('is_available', CategoryFilter)
    list_select_related = ('category', 'category__restaurant',)
```



```

search_fields = ('name', 'description')
filter_horizontal = ('allergens',)

def get_queryset(self, request):
    return super().get_queryset(request).prefetch_related('allergens')

```

And now we got our JOIN query on the allergens table (Figure 8-9).

```

SELECT ... FROM "backoffice_allergen" INNER JOIN "backoffice_dish_allergens" ON ("backoffice_allergen"."id" =
"backoffice_dish_allergens"."allergen_id") WHERE "backoffice_dish_allergens"."dish_id" IN (4129, 3724, 1699, 1006, 479, 3995,
1964, 368, 3333, 2905, 2377, 1519, 1483, 3647, 534, 3157, 2834, 1915, 360, 3748, 3729, 3696, 1799, 743, 439, 3274, 3257, 2107,
1750, 4037, 2702, 2111, 1235, 630, 560, 3873, 3734, 3362, 1745, 1543, 1466, 1271, 1123, 2808, 2154, 1347, 3977, 3947, 3828,
2888, 1965, 1295, 904, 328, 4148, 2567, 1468, 1141, 4227, 3036, 2029, 1204, 3825, 3709, 2761, 2753, 2307, 1903, 1844, 321,
3024, 1514, 842, 817, 2042, 1403, 483, 4261, 2999, 2856, 2349, 3221, 3085, 1278, 4158, 3392, 2817, 1473, 1216, 869, 4309,
4294, 4293, 3201, 3144, 3081, 1847, 934, 600, 3379)

```

SQL queries from 1 connection

Figure 8-9. Prefetching many-to-many relations via `get_queryset().prefetch_related()` (allergens)

get_queryset method is a default Django Admin method that returns the queryset of the model. We didn't learn it yet, but most of **Django Admin methods** can be overridden to customize the behavior of the admin interface. And yes you are right, we could have avoided to use the `list_select_related` attribute and use it in the queryset instead. The results would be the same.

```

def get_queryset(self, request):
    return super().get_queryset(request).prefetch_related('allergens')
        .select_related('category', 'category__restaurant')

```

Simple Caching Strategies

Implement caching for expensive computations is a great way to improve performance. We will see later in this book how to implement robust caching with a tool like **Redis**, but for now, we can focus on using simple cache memory.

Let's say we want to display in our RestaurantAdmin the total number of orders for each restaurant. First, let's add an order using our admin interface (Figure 8-10).

The screenshot shows a web application titled "Uber Eat like Admin". The top navigation bar includes "Home", "Backoffice", "Orders", and "Add order". The right side of the header says "WELCOME, CSURBIER. CHANGE PASSWORD / LOG OUT". The left sidebar has two main sections: "AUTHENTICATION AND AUTHORIZATION" with "Groups" and "Users" (both with "+ Add" links), and "BACKOFFICE" with "Allergens", "Categories", "Dishes", "Order Items", "Orders" (highlighted in yellow with "+ Add"), and "Restaurants" (with "+ Add"). The main content area is titled "Add order" and contains the following form fields: "Customer:" with a dropdown menu showing "csurbier", "Restaurant:" with a dropdown menu showing "Byrd Ltd from Holtshire", "Status:" with a dropdown menu showing "Pending", and "Delivery address:" with a text input field containing "my house". Below these fields is a "Total amount:" field showing "12". At the bottom of the form are three buttons: "SAVE", "Save and add another", and "Save and continue editing".

Figure 8-10. *Creating an Order in admin to generate data for a “total orders” list_display example*

And then to display the number of orders (see Figure 8-11), we can write a custom method as we learned previously and add it to the `list_display` attribute:

```
from django.core.cache import cache
from django.utils.functional import cached_property

class RestaurantAdmin(admin.ModelAdmin):
    @admin.display(description="Total Orders")
    def total_orders(self, obj):
        total = obj.orders.count()
        return total

list_display = ('colored_name', 'total_orders', 'location', 'ownerEmail', 'full_address', 'phone', 'city', 'is_active', 'created_at')
```

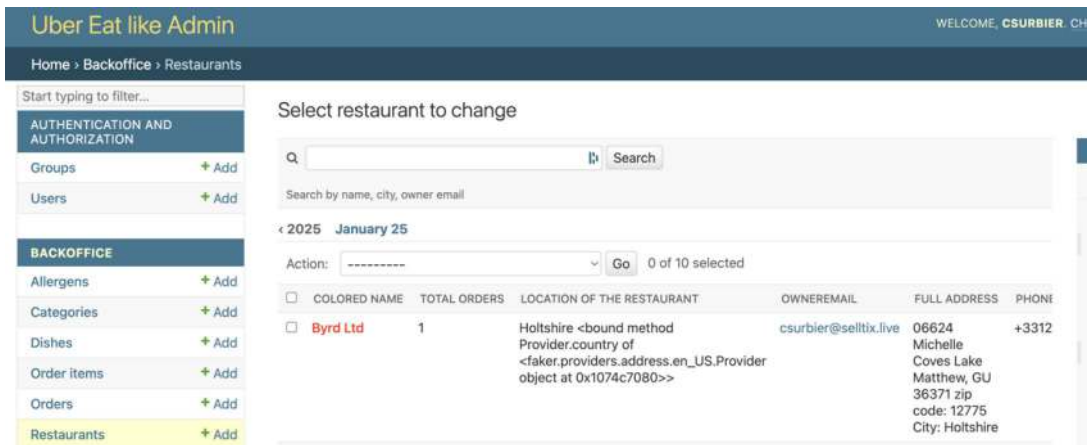


Figure 8-11. Displaying a computed “Total Orders” column in `list_display` (candidate for caching)

Now let’s imagine we have thousands of orders in our database. To avoid calculating the total number of orders for each restaurant on each page load, we can use the cache method from `django.core.cache` import `cache` to cache the result of the `total_orders` method:

```
@admin.display(description="Total Orders")
def total_orders(self, obj):
    cache_key = f'restaurant_orders_{obj.pk}'
    total = cache.get(cache_key)

    if total is None:
        print("===Calculating total orders===")
        total = obj.orders.count()
        cache.set(cache_key, total, 3600) # Cache for 1 hour
    else:
        print("===Using cached total orders===")
    return total
```

Let’s reload our page once again; we should see in the console (as many lines as we have results)

```
===Calculating total orders===
===Calculating total orders===
===Calculating total orders===
===Calculating total orders===
...
```

Then reload the page and we should see

```
===Using cached total orders===  
===Using cached total orders===  
===Using cached total orders===  
===Using cached total orders===  
...
```

This cache instruction will cache the result in the memory of the server and will be deleted after a certain amount of time. So it's important to use it wisely. As said in introduction, to implement a robust caching system, we should use a tool like **Redis** or **Memcached**.

Pagination Handling

A simple way to improve the Django Admin performance is to control the number of objects displayed per page and implement an efficient pagination. Controlling pagination will improve performance in several key ways:

- Reduces database load by limiting the number of records fetched at once
- Minimizes memory usage on the server since fewer objects need to be instantiated
- Speeds up page rendering time in the browser
- Decreases network bandwidth usage by transferring smaller chunks of data
- Prevents timeouts when dealing with large datasets

To do so, we can use the `list_per_page` attribute, which allows us to specify the number of objects to display per page. The `list_max_show_all` attribute allows us to specify the maximum number of objects to display when showing all objects, and the `show_full_result_count` attribute allows us to disable the full count for better performance:

```
@admin.register(Order)
class RestaurantAdmin(admin.ModelAdmin):
    list_per_page = 10 # Default number of objects per page
    list_max_show_all = 200 # Maximum number when showing all
    show_full_result_count = False # Disable full count for better
performance
```

Basic Database Indexes

Another quite simple way to optimize performances is to add indexes to frequently searched or filtered fields in your **models**. For our Restaurant model, we can add the following indexes:

```
class Restaurant(models.Model):
    name = models.CharField(_('name'), max_length=200, db_index=True)
    city = models.CharField(_('city'), max_length=100, db_index=True)
    created_at = models.DateTimeField(auto_now_add=True, db_index=True)

    class Meta:
        indexes = [
            models.Index(fields=['name', 'city']),
            models.Index(fields=['-created_at']),
        ]
```

We use the `db_index=True` attribute to create an index on the fields.

Let's run the makemigrations command:

```
python manage.py makemigrations
Migrations for 'backoffice':
  backoffice/migrations/0003_alter_restaurant_city_alter_restaurant_
created_at_and_more.py
  ~ Alter field city on restaurant
  ~ Alter field created_at on restaurant
  ~ Alter field name on restaurant
```

```
+ Create index backoffice__name_023dfc_idx on field(s) name, city of
model restaurant
+ Create index backoffice__created_321a61_idx on field(s) -created_at
of model restaurant
```

and the migrate command:

```
python manage.py migrate
```

Operations to perform:

Apply all migrations: admin, auth, backoffice, contenttypes, sessions

Running migrations:

```
Applying backoffice.0003_alter_restaurant_city_alter_restaurant_created_
at_and_more... OK
```

These optimizations can significantly improve the performance of our Django Admin interface, especially when dealing with large datasets or complex relationships.

Summary

In this **part**, we moved from “default admin” to a more usable backoffice by customizing how models are presented and managed. We explored the essential `ModelAdmin` options that shape the day-to-day experience in Django Admin—list pages (columns, links, pagination, ordering), filtering and searching, and how to improve clarity with computed fields and small display tweaks.

We also introduced more advanced patterns you’ll use in real projects, such as custom list filters, queryset optimization with `select_related()`/`prefetch_related()` mandatory for performance, and basic site branding. With these building blocks, you can make the admin faster, clearer, and better adapted to your team’s workflow.

In the next chapter, we will build upon these fundamentals to explore advanced customization techniques.

Here is our `admin.py` file for now:

```
from django.contrib import admin
from django.http import HttpResponse
from .models import Allergen, Restaurant, Category, Dish, Order, OrderItem
from django.contrib.auth.models import User
```

```

from django.utils.translation import gettext_lazy as _ # Use this to get
translated strings
import csv
from django import forms
from django.core.cache import cache

```

```

class RestaurantOwnerFilter(admin.SimpleListFilter):
    title = 'Owner Type'
    parameter_name = 'owner_type'

    def lookups(self, request, model_admin):
        if not request.user.is_superuser:
            return ()
        return (
            ('with_owner', _('Has Owner')),
            ('without_owner', _('No Owner')),
        )

    def queryset(self, request, queryset):
        if self.value() == 'with_owner':
            return queryset.exclude(owner=None)
        if self.value() == 'without_owner':
            return queryset.filter(owner=None)

class RestaurantWithoutPhoneFilter(admin.SimpleListFilter):
    title = 'Restaurant without phone'
    parameter_name = 'phone_filter'

    def lookups(self, request, model_admin):
        qs = model_admin.get_queryset(request)
        if qs.filter(
            phone=''
        ).exists():
            yield ("no_phone", _("No phone set"))

    def queryset(self, request, queryset):
        if self.value() == 'no_phone':
            return queryset.filter(phone='')
        return queryset

```

```

class CategoryInline(admin.TabularInline):
    model = Category
    extra = 4
    show_change_link = True

class RestaurantForm(forms.ModelForm):
    confirm_phone = forms.CharField(max_length=20)

    class Meta:
        model = Restaurant
        fields = '__all__'

    def clean_phone(self):
        phone = self.cleaned_data.get('phone')
        confirm_phone = self.cleaned_data.get('confirm_phone')
        if not phone.startswith('+237'):
            raise forms.ValidationError("Phone number must start
                with +237")

        if phone != confirm_phone:
            raise forms.ValidationError("Phone numbers must match")
        return phone
    """

    def clean(self):
        cleaned_data = super().clean()
        phone = cleaned_data.get('phone')
        confirm_phone = cleaned_data.get('confirm_phone')

        if phone != confirm_phone:
            raise ValidationError("Phone numbers must match")
        return cleaned_data
    """

@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    form = RestaurantForm
    actions = ['make_inactive', 'export_to_csv']

```

```

@admin.action(
    description="Mark selected restaurants as inactive"
)
def make_inactive(self, request, queryset):
    updated = queryset.update(is_active=False)
    self.message_user(
        request,
        f"{updated} restaurants were successfully marked as inactive."
    )
@admin.action(description="Export selected restaurants to CSV")
def export_to_csv(self, request, queryset):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = 'attachment;
    filename="restaurants.csv"'

    writer = csv.writer(response)
    writer.writerow(['Name', 'Address', 'Phone', 'Owner'])

    for restaurant in queryset:
        writer.writerow([
            restaurant.name,
            restaurant.address,
            restaurant.phone,
            restaurant.owner.email
        ])

    return response

#inlines = [CategoryInline]
@admin.display(description="Full address",ordering="address")
def full_address(self,obj):
    return f"{obj.address} zip code: {obj.zip_code} City: {obj.city}"

@admin.display(ordering="owner__email")
def ownerEmail(self, obj):
    return obj.owner.email

```



```

@admin.display(description="Total Orders")
def total_orders(self, obj):
    cache_key = f'restaurant_orders_{obj.pk}'
    total = cache.get(cache_key)

    if total is None:
        print("===Calculating total orders===")
        total = obj.orders.count()
        cache.set(cache_key, total, 3600) # Cache for 1 hour
    else:
        print("===Using cached total orders===")
    return total

fieldsets = (
    ("My first section", {
        'fields': ('name', 'image', ('address', 'zip_code', 'city'),
        'country', 'phone', 'confirm_phone', ),
        'description': 'This is the first section of the form.'
    }),
    ('Advanced options', {
        'classes': ('collapse',),
        'fields': ('is_active', 'owner',),
    }),
)

#readonly_fields = ('phone',)
list_display = ('colored_name', 'total_orders', 'location', 'ownerEmail', '
full_address', 'phone', 'city', 'is_active', 'created_at')
list_select_related = ('owner',)
list_display_links = ('colored_name', 'ownerEmail',)
#list_filter = (RestaurantOwnerFilter, RestaurantWithoutPhoneFilter, '
is_active', 'city',)
list_filter = ( ("phone", admin.EmptyFieldListFilter), RestaurantOwner
Filter, 'is_active', 'city',)
search_fields = ('name__exact', 'city', 'owner_email',)
search_help_text = _("Search by name, city, owner email")
#list_editable = ('name',)

```

```

list_per_page = 10 # Default number of objects per page
list_max_show_all = 200 # Maximum number when showing all
show_full_result_count = False # Disable full count for better
performance
date_hierarchy = 'created_at'
empty_value_display = "-empty-"

@admin.register(Allergen)
class AllergenAdmin(admin.ModelAdmin):

    list_display = ('name', 'description', )
    search_fields = ('name',)

@admin.register(Category)
class CategoryAdmin(admin.ModelAdmin):

    list_display = ('name', 'restaurant', 'is_active')
    list_filter = ('is_active', 'restaurant',)
    search_fields = ('name',)
    list_select_related = ('restaurant', 'restaurant__owner',)

class CategoryFilter(admin.SimpleListFilter):
    title = 'category'
    parameter_name = 'category'

    def lookups(self, request, model_admin):
        categories = Category.objects.select_related('restaurant').all()
        return [(c.id, str(c)) for c in categories]

    def queryset(self, request, queryset):
        if self.value():
            return queryset.filter(category_id=self.value())
        return queryset

@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'price', 'is_available',)
    list_filter = ('is_available', CategoryFilter)
    #list_select_related = ('category', 'category__restaurant',)

```

```

search_fields = ('name', 'description')
filter_horizontal = ('allergens',)

def get_queryset(self, request):
    return super().get_queryset(request).prefetch_related('allergens').
        select_related('category', 'category__restaurant')

@admin.register(Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ('id', 'customer', 'restaurant', 'status', 'total_
amount', 'created_at')
    list_filter = ('status', )
    search_fields = ('customer__username', 'restaurant__name')
    list_select_related = ('customer', 'restaurant',)
    date_hierarchy = 'created_at'

@admin.register(OrderItem)
class OrderItemAdmin(admin.ModelAdmin):
    list_display = ('order', 'dish', 'quantity', 'unit_price')
    list_filter = ('order__status',)
    search_fields = ('dish__name',)
    list_select_related = ('order', 'dish',)

# Site branding and appearance
admin.site.site_header = "Uber Eat like Admin"
admin.site.index_title = "Welcome back to Uber Eat like Admin Portal"
admin.site.site_url = None # Remove "View Site" link
admin.site.enable_nav_sidebar = True # Enable the navigation sidebar

```

PART III

Advanced Customization

This part will take us deeper into the advanced customization capabilities of Django Admin. We'll learn how to extend our admin interface beyond its core functionality to match business requirements or workflows.

We will focus on the following topics:

- **Extend Features:** Push Django Admin's built-in features to their limits, creating sophisticated list displays, advanced filters, and implementing complex validation logic.
- **Template System Inheritance:** Learn how to completely transform the admin interface by understanding Django's template inheritance system and implementing custom tags and context processors.
- **Custom Views and URLs:** Move beyond standard model views by creating custom pages, dashboards, and specialized workflows within the admin interface.
- **REST API Integration with Django Admin:** Connect your admin interface with external systems and APIs, enabling powerful integration capabilities.
- **Background Tasks:** Implement Celery for handling resource-intensive operations, allowing your admin interface to manage long-running processes efficiently.
- **Advanced Query Optimization:** Perform raw SQL queries in Django Admin for handling complex relationships.

CHAPTER 9

Extend Features

list_display

We saw in the previous chapter how to customize the list display of a model in the admin interface. Let's explore some more advanced features.

list_display: Display Custom Link

Django Admin allows us to add clickable links or buttons to our list display, enabling quick navigation to related views or actions. This is particularly useful for providing shortcuts to related data or initiating specific operations.

Let's implement a custom link in our Restaurant model to view all orders associated with a restaurant:

```
from django.utils.html import format_html
```

First, we need to import the `format_html` function that we already learned in a previous chapter.

Then we will add a custom method which will return a link to the orders page for the selected restaurant. The `@admin.display` decorator configures how a `ModelAdmin` method appears in the admin (e.g., as a changelist column when the method is included in `list_display`), including its column label and sort behavior:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    # ...

    @admin.display(description="View Orders")
    def view_orders(self, obj):
```

```

return format_html(
    '<a class="button" href="/admin/backoffice/order/?restaurant__'
    'id__exact={}" target="_blank">Orders</a>',
    obj.id
)

list_display = ('colored_name', 'total_orders', 'view_orders', 'location',
               'ownerEmail', 'full_address', 'phone', 'city', 'is_active',
               'created_at')

```

This creates a “View Orders” column in the restaurant list (see Figure 9-1) with a button for each restaurant. When clicked, the button opens a filtered view of the Order admin page showing only orders for that specific restaurant.

Key points about custom links in `list_display`:

1. We use the `format_html()` function to safely render HTML in the admin interface, meaning it renders real HTML (links, `<strong>`, `<em>`, etc.), but anything that comes from the database or user input is escaped so it cannot be interpreted as active HTML/JavaScript. It’s a good practice to use this function to avoid security issues.
2. The `@admin.display(description=“View Orders”)` decorator sets the column header text.
3. The URL uses Django Admin’s filter syntax (`restaurant__id__exact`) to filter the orders.
4. The `target="_blank"` attribute opens the link in a new tab.

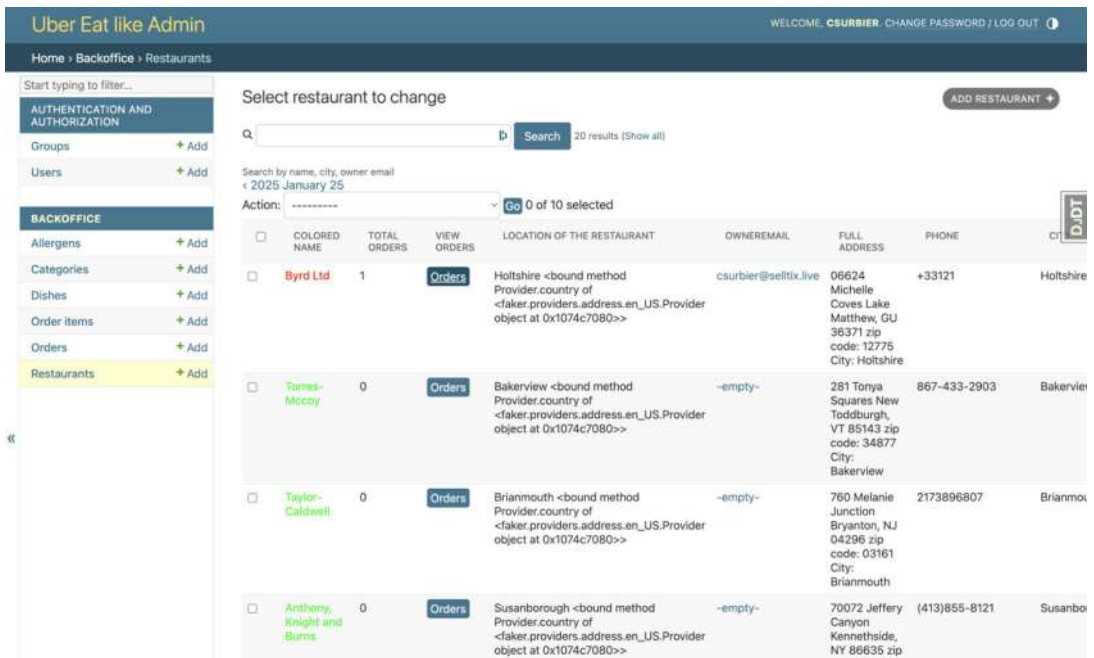


Figure 9-1. Custom `list_display` link button to open a filtered `Orders` changelist for a restaurant

We can extend this pattern to create links to any view in your admin interface, to external systems, or to implement quick actions without leaving the list view.

As another example, we could add a link to view a restaurant's details on a map (see Figure 9-2 and 9-3):

```
@admin.display(description="Map Location")
def view_on_map(self, obj):
    return format_html(
        '<a class="button" href="https://maps.google.com/?q={},{},{}"
        target="_blank">'
        '<i class="fas fa-map-marker"></i> Map</a>',
        obj.address,
        obj.city
    )

list_display = ('colored_name', 'total_orders', 'view_orders', 'view_on_map',
                'ownerEmail', 'full_address', 'phone', 'city', 'is_active',
                'created_at')
```

I removed the 'location' field and replaced it with the view_on_map method.

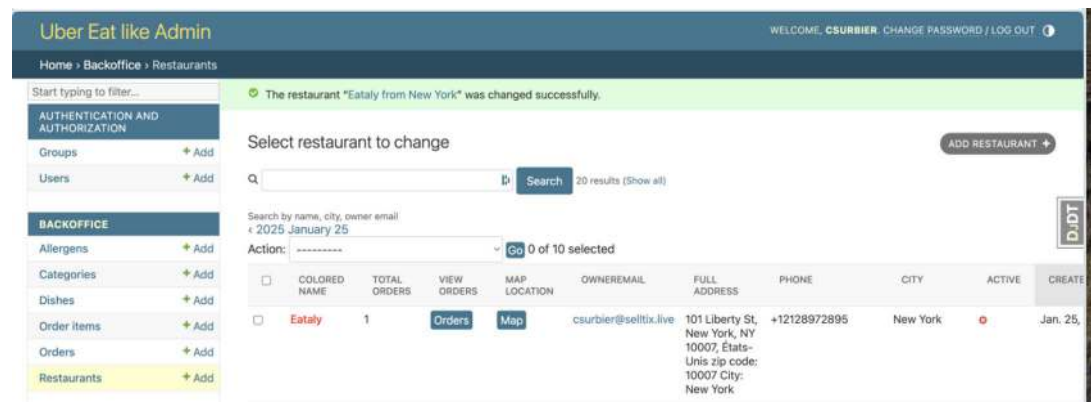


Figure 9-2. Custom `list_display` link opening Google Maps for a restaurant address

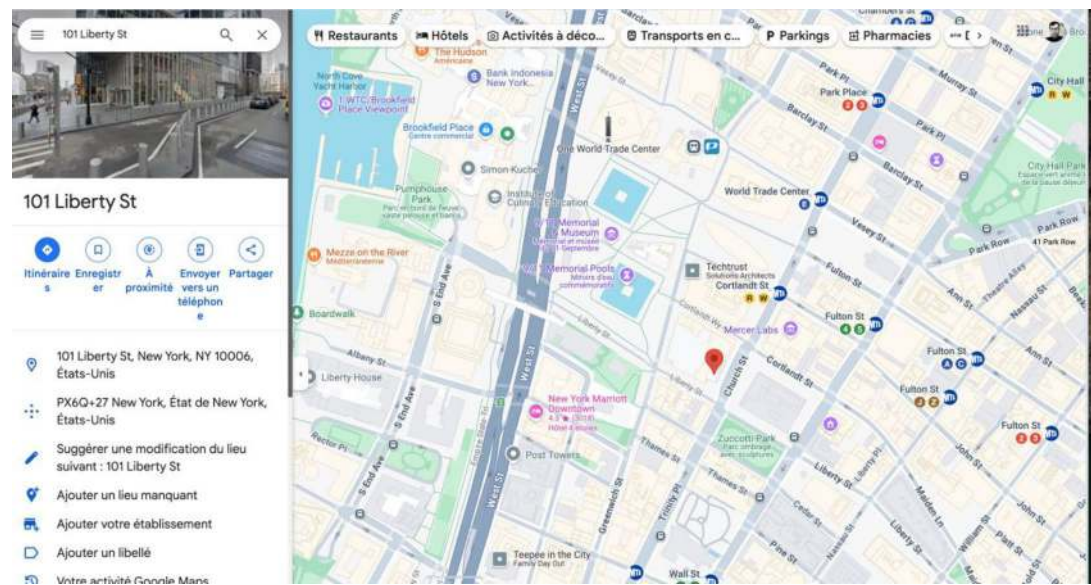


Figure 9-3. Example map view opened from the custom “Map Location” link

I have also edited my restaurant data to update with a real address. Remember, the data generated earlier was fake.

list_display: Custom Admin URLs and Actions

When building advanced Django Admin interfaces, we'll often need to implement custom actions beyond the standard CRUD operations. We are talking about actions directly usable from the list view and not using the [actions] dropdown, as we learned earlier.

To do that, we need to explore how Django's URL system connects actions in our interface with custom view functions.

To illustrate that, let's add a button to activate (see Figure 9-4) or deactivate (see Figure 9-5) a restaurant and then display to the user a message.

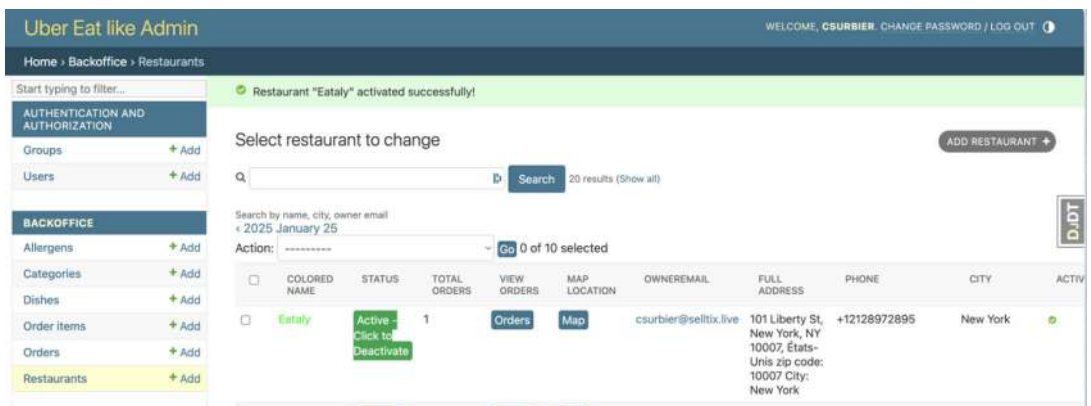


Figure 9-4. Custom `list_display` button to toggle a restaurant active/inactive status

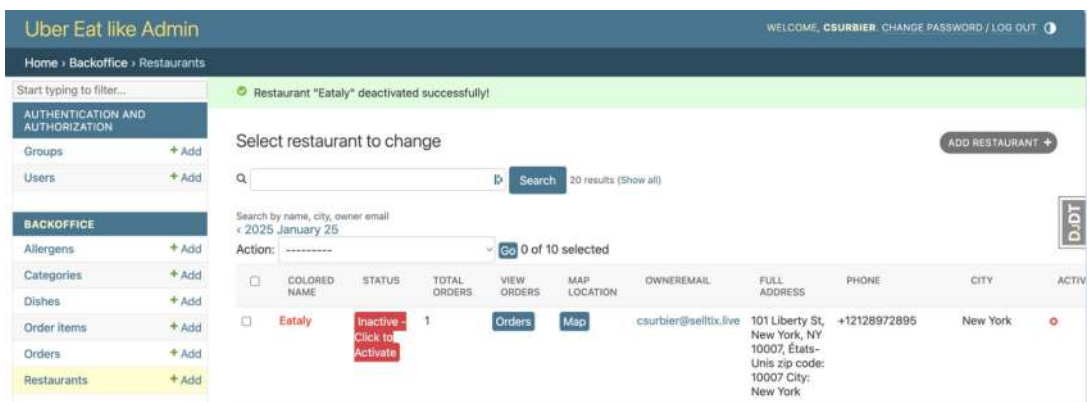


Figure 9-5. Restaurant list after toggling status (button label and color updated)

Let's start by adding the import we will need:

```
from django.http import HttpResponseRedirect
from django.urls import path, reverse
from django.utils.html import format_html
from django.contrib import messages
```

Then we will implement the `admin.decorator` that will display the button in our list view and handle the click event:

```
@admin.display(description="Status")
def toggle_active_button(self, obj):
    url = reverse('admin:toggle-active-restaurant', args=[obj.pk])

    return format_html(
        '<a class="button" style="background-color: {}; color: white;"
        href="{}">{}</a>',
        '#28a745' if obj.is_active else '#dc3545',
        url,
        'Active - Click to Deactivate' if obj.is_active else 'Inactive -
        Click to Activate'
    )
```

The line `url = reverse('admin:toggle-active-restaurant', args=[obj.pk])` is an example of Django's URL reversal system. The `reverse()` function converts a URL name into an actual URL path. In the admin context, the name follows this pattern:

- **admin:** prefix indicates that this is an admin URL. It's the namespace for all admin URLs.
- **toggle-active-restaurant** is the name we gave to our custom URL pattern.

The `args=[obj.pk]` provides the restaurant ID as a parameter to the URL pattern.

For the `reverse()` function to work, we need to declare the URL pattern in our `ModelAdmin` class. This is done by overriding the default Django Admin `get_urls()` method:

```
def get_urls(self):
    urls = super().get_urls()
```

```

my_urls = [
    path('toggle-active/<path:object_id>/',
        self.admin_site.admin_view(self.toggle_active),
        name='toggle-active-restaurant'),
]
return my_urls + urls

```

This method

1. Gets the default admin URLs with `super().get_urls()`
2. Defines our custom URL pattern that captures an `object_id` parameter (our restaurant ID)
3. Associates this URL with a **`toggle_active`** view method
4. The URL pattern a name that we can use with `reverse()`
5. Returns the combined URLs, with our custom ones taking precedence

The **`self.admin_site.admin_view()`** wrapper is crucial as it applies all the admin's authentication and permission checks to our custom view. Now, we need to implement the **`toggle_active`** view method:

```

def toggle_active(self, request, object_id):
    """Handle the URL request for toggling active status"""
    restaurant = self.get_object(request, object_id)
    restaurant.is_active = not restaurant.is_active
    restaurant.save()

    status = "activated" if restaurant.is_active else "deactivated"
    messages.success(request, f'Restaurant "{restaurant.name}" {status}
    successfully!')

    return HttpResponseRedirect("../../")

```

Warning In this example, we are doing one read and one write. If two staff members click “toggle” at almost the same time, both can read the same value, each negates it in memory, and both save—you can end up with no net change (classic lost update/race). For production at scale, you might use Queryset.
`update(F('is_active')=...):`

```
from django.db.models import F
Restaurant.objects.filter(pk=object_id).update(is_active=~F("is_active"))
```

The **toggle_active** method will handle the actual business logic when a user clicks the button:

- Uses `self.get_object()` to retrieve the restaurant instance
- Toggles its active status
- Saves the change to the database
- Displays a success message to the user using Django’s messages framework (`messages.success()`)
- Redirects back to the restaurant list view

So to resume the entire workflow:

- User sees a button in the admin list view (rendered by `toggle_active_button`).
- User clicks the button, triggering a request to our custom URL.
- Django routes this request to our `toggle_active` method.
- The method updates the restaurant’s status and redirects back to the list view.
- User sees the updated status and a confirmation message.

This pattern is incredibly powerful for adding custom functionality to our Django Admin interface. We can use it to implement any action that needs to be performed on our models directly from the list view, without requiring users to navigate to separate pages.

The button changes color (green for active, red for inactive) and text based on the current status, providing clear visual feedback to the user.

HTTP semantics: In the HTTP model, **GET** requests are expected to be *safe* (they should not change server state by themselves). Toggling a field in the database is a state change, so some guides recommend issuing a **POST** (often with a CSRF token) instead. This chapter uses a **GET** link to keep the code short. The custom view is still wrapped with **admin_view**, so only authenticated staff with the right model permissions can run it. For many private admin UIs, that is enough. You can use a **POST** form and CSRF if you need stricter alignment with the safe-GET rule or extra protection.

list_display: Chain Filter

In the previous chapter, we learned to implement custom filters in Django Admin. Let's now learn how to implement dependent or cascading filters in Django Admin. Cascading filters are filters that depend on the selection of another filter. This powerful feature helps users navigate complex data relationships.

Before that, let's take a moment to discuss about the Category model:

```
class Category(models.Model):
    name = models.CharField(_('name'), max_length=100)
    picto = models.ImageField(upload_to='restaurants/', blank=True,
                             null=True)
    restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE,
                                   related_name='categories')
    is_active = models.BooleanField(_('active'), default=True)
    created_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        verbose_name = _('category')
        verbose_name_plural = _('categories')
        ordering = ['name']

    def __str__(self):
        return f"{self.restaurant.name} - {self.name}"
```

For now, a Category is linked to a Restaurant. With this design, each category belongs to exactly one restaurant. This means if 50 restaurants want a “Desserts” category, we need 50 separate category records in the database.

Another approach could be to use a many-to-many relationship between Restaurant and Category. This would allow a restaurant to have multiple categories and a category to belong to multiple restaurants:

```
class Category(models.Model):
    name = models.CharField(_('name'), max_length=100)
    picto = models.ImageField(upload_to='categories/', blank=True,
                              null=True)
    restaurants = models.ManyToManyField(Restaurant, related_
name='categories')
    is_active = models.BooleanField(_('active'), default=True)
    created_at = models.DateTimeField(auto_now_add=True)

class Meta:
    verbose_name = _('category')
    verbose_name_plural = _('categories')
    ordering = ['name']

def __str__(self):
    return self.name # Changed since a category can belong to multiple
restaurants
```

Advantages of many-to-many:

- **Reusability:** Common categories like “Appetizers” or “Desserts” can be shared.
- **Consistency:** Category names remain consistent across restaurants.
- **Maintainability:** Updating a category name updates it for all restaurants.

Disadvantages of many-to-many:

- **Less Customization:** Restaurants can’t customize category names to their branding.

- **Ordering Issues:** Different restaurants might want different ordering of categories.
- **Complexity:** Admin interface becomes more complex to manage.

Let's modify our models.py to use the many-to-many relationship and then run the migrations:

```
python manage.py makemigrations
```

```
Migrations for 'backoffice':
```

```
backoffice/migrations/0004_remove_category_restaurant_category_
restaurants_and_more.py
```

```
- Remove field restaurant from category
```

```
+ Add field restaurants to category
```

```
~ Alter field picto on category
```

```
python manage.py migrate
```

Operations to perform:

Apply all migrations: admin, auth, backoffice, contenttypes, sessions

Running migrations:

Applying backoffice.0004_remove_category_restaurant_category_restaurants_and_more... OK

We need to update our admin.py file to reflect the new many-to-many relationship and remove the restaurant field from the CategoryAdmin class:

```
@admin.register(Category)
```

```
class CategoryAdmin(admin.ModelAdmin):
```

```
    list_display = ('name', 'is_active')
```

```
    list_filter = ('is_active',)
```

```
    search_fields = ('name',)
```

We also need to update the Dish models to set to which restaurant the dish belongs to:

```
class Dish(models.Model):
```

```
    name = models.CharField(_('name'), max_length=200)
```

```
    description = models.TextField(_('description'))
```

```
    price = models.DecimalField(_('price'), max_digits=6, decimal_places=2)
```

```

category = models.ForeignKey(Category, on_delete=models.CASCADE,
related_name='dishes_category')
restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE,
related_name='dishes_restaurant')
allergens = models.ManyToManyField(Allergen, blank=True, related_
name='dishes')
is_available = models.BooleanField(_('available'), default=True)
image = models.ImageField(_('image'), upload_to='dishes/', blank=True)

class Meta:
    verbose_name = _('dish')
    verbose_name_plural = _('dishes')
    ordering = ['name']

def __str__(self):
    return f"{self.name}"

```

But now, if we try to run the migrations, we will get the following error:

```
python manage.py makemigrations
```

It is impossible to add a non-nullable field 'restaurant' to dish 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:

This is because we have a non-nullable field in our Dish model that is not set to a default value. To fix this, we need to set a default value for the restaurant field in the Dish model. Something like this:

```

class Dish(models.Model):
    name = models.CharField(_('name'), max_length=200)
    description = models.TextField(_('description'))
    price = models.DecimalField(_('price'), max_digits=6, decimal_places=2)

```



```

category = models.ForeignKey(Category, on_delete=models.CASCADE,
related_name='dishes_category')
restaurant = models.ForeignKey(Restaurant, on_delete=models.CASCADE,
                               elated_name='dishes_restaurant',
                               default=1)

allergens = models.ManyToManyField(Allergen, blank=True, related_
name='dishes')
is_available = models.BooleanField(_('available'), default=True)
image = models.ImageField(_('image'), upload_to='dishes/', blank=True)

class Meta:
    verbose_name = _('dish')
    verbose_name_plural = _('dishes')
    ordering = ['name']

def __str__(self):
    return f"{self.name}"

```

But doing this means that we are setting the restaurant to “the first row” in the Restaurant table: `default=1` means “use the Restaurant whose **primary key is exactly 1**.” If no row has that key (e.g., the first Restaurant was created with `id=2` after a failed insert), the migration or inserts can **fail** or your dishes can end up **pointing at a restaurant you did not intend**. Worse, every existing Dish that needed a default during the migration is forced to the **same** restaurant, which is rarely correct. In a real app, we would handle this with a **data migration** (set restaurant per row from your business rules) or a **nullable** field until data is backfilled, not a hard-coded 1 in the model.

A **separate** design topic is that we have been using **integer** primary keys. They are **predictable** in public URLs or API paths: `/restaurant/1/`, `/restaurant/2/`, and so on are easy to script; for an attacker, it’s easy to enumerate the number of records in the database. So sequential integers make **metadata leakage** easier (e.g., inferring how many users or orders you have from the latest id), which can matter in some product or compliance contexts.

So even if security should be based on the authentication and permissions (**check that the caller is allowed to read or change that row**) and not obscurity, I prefer to use UUIDs for primary keys which can’t be guessed: in **defense in depth**, opaque identifiers can help, if a route is misconfigured or someone forgets a permission check. As I said, an attacker can **enumerate** low integers far more easily than trillions of valid UUIDs.

So we are going to use the UUID as the primary key. Django can generate it in Python (`uuid.uuid4`); the database stores a fixed-length unique value.

Note UUID PKs trade a bit of space and some index locality compared to sequential integers; for most projects, this is acceptable, but it's a design tradeoff.

Here's how to implement UUID primary keys in our models:

```
import uuid
from django.db import models

class Restaurant(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,
        editable=False)
    # other fields remain the same
    name = models.CharField(_('name'), max_length=200)
    # ...
```

We need to add the following line to each class of our models:

```
id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
```

We could run the migrations because we are using the **SQLite** database. But if we were using a different database, we would get an error when trying to change primary keys from **integers** to **UUID** fields.

Also, the data we generated earlier are not relevant since we modified our models to introduce **many to many relationships** in our Category model and a **foreign key** to Restaurant in our Dish model.

So, we are going to delete our database and start from scratch again then modify our script which generate random data to generate data that matches our new models.

Use PostgreSQL

I will use PostgreSQL as my database for this project, but you can use any other database you want with Django Admin (which supports UUIDs as primary keys if you choose to use them).

To use PostgreSQL, we need to install the `psycopg2` package and the `python-dotenv` packages:

- `psycopg2` (or the binary wheel) is what Django needs to connect to PostgreSQL.
- `python-dotenv` is optional: we will use it so `settings.py` can read database credentials and host information from a `.env` file instead of hard-coding them. You could put the same values in the real environment (or in another secret mechanism) and skip `python-dotenv` entirely.

So let's add them to our `requirements.txt` file:

```
psycopg2-binary==2.9.9
python-dotenv==1.0.0
```

Then, we are going to modify our `settings.py` file to use PostgreSQL as our database and import the environment variables from our `.env` file:

```
from dotenv import load_dotenv
# Build paths inside the project like this: BASE_DIR / 'subdir'.
# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent
# Load environment variables
load_dotenv(BASE_DIR / '.env')

# ...
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.getenv('POSTGRESQL_ADDON_DB', 'djangoadmin'),
        'USER': os.getenv('POSTGRESQL_ADDON_USER', 'postgres'),
        'PASSWORD': os.getenv('POSTGRESQL_ADDON_PASSWORD', ''),
```

```
'HOST': os.getenv('POSTGRESQL_ADDON_HOST', 'localhost'),
'PORT': os.getenv('POSTGRESQL_ADDON_PORT', '5432'),
}
```

You need to have a PostgreSQL database running on your local machine or using a cloud provider such as Digital Ocean.

Now, we need to remove all the existing migrations files:

```
rm -rf ./backoffice/migrations/*
```

Warning Be careful with this command. A mistyped command could result in the loss of your data. You can remove the `-rf` option if you want to be more careful and want a confirmation before each file is deleted.

And run the migrations again:

```
python manage.py makemigrations backoffice
python manage.py migrate
```

And recreate our superuser:

```
python manage.py createsuperuser
```

Create Fake Data (Again)

We will update our script to generate random data that matches our new models and with some realistic data this time:

```
import os
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "djangoadmin.settings")
import django
django.setup()

from faker import Faker
```

```

from backoffice.models import Restaurant, Category, Dish, Allergen, Order,
OrderItem
from model_bakery.recipe import Recipe, foreign_key
from django.contrib.auth.models import User
from decimal import Decimal
import random

fake = Faker()

# Create common restaurant categories
RESTAURANT_CATEGORIES = ['Italian', 'Japanese', 'Mexican', 'Indian', 'French']
FOOD_BY_CATEGORY = {
    'Italian': ['Pizza Margherita', 'Pasta Carbonara', 'Risotto', 'Tiramisu'],
    'Japanese': ['Sushi Roll', 'Ramen', 'Tempura', 'Miso Soup'],
    'Mexican': ['Tacos', 'Enchiladas', 'Guacamole', 'Quesadillas'],
    'Indian': ['Butter Chicken', 'Biryani', 'Naan', 'Tikka Masala'],
    'French': ['Coq au Vin', 'Ratatouille', 'Croissant', 'Crème Brûlée']
}

# Create allergens first
COMMON_ALLERGENS = [
    ('Peanuts', 'May contain traces of peanuts'),
    ('Gluten', 'Contains gluten - wheat based'),
    ('Lactose', 'Contains dairy products'),
    ('Shellfish', 'Contains shellfish'),
    ('Eggs', 'Contains eggs')
]

allergens = [Allergen.objects.create(name=name, description=desc) for name,
desc in COMMON_ALLERGENS]

# Create users with realistic profiles
users = [User.objects.create_user(
    username=fake.user_name(),
    email=fake.email(),
    password='password123',
    first_name=fake.first_name(),
    last_name=fake.last_name())

```

```

) for _ in range(5)]

# Create restaurants with coherent data
for user in users:
    for _ in range(2):
        restaurant = Restaurant.objects.create(
            name=f"{fake.company()} {random.choice(['Restaurant', 'Bistro',
            'Café'])}",
            address=fake.street_address(),
            zip_code=fake.postcode()[:20], # Limit zip_code to 20
            characters
            city=fake.city(),
            country=fake.country(),
            phone=fake.phone_number()[:20], # Limit phone to 20 characters
            owner=user
        )

# Create categories based on cuisine type
for cuisine in random.sample(RESTAURANT_CATEGORIES, 3):
    category = Category.objects.create(
        name=cuisine
    )
    # Set the many-to-many relationship properly
    category.restaurants.set([restaurant])

# Create dishes specific to the cuisine
for dish_name in FOOD_BY_CATEGORY[cuisine]:
    dish = Dish.objects.create(
        name=dish_name,
        description=fake.paragraph(),
        price=Decimal(random.uniform(8.99, 29.99)).
        quantize(Decimal('0.01')),
        category=category,
        restaurant=restaurant,
        is_available=random.choice([True, True, True,
        False]) # 75% chance of being available
    )

```

```

        # Add random allergens to dishes
        dish.allergens.set(random.sample(allergens, k=random.
            randint(0, 3)))

# Create some orders
for user in users:
    for restaurant in Restaurant.objects.all():
        # Create 1-3 orders per user per restaurant
        for _ in range(random.randint(1, 3)):
            order = Order.objects.create(
                customer=user,
                restaurant=restaurant,
                status=random.choice(['pending', 'confirmed', 'preparing',
                    'delivered']),
                delivery_address=fake.address(),
                total_amount=Decimal('0')
            )

            # Add 1-5 items to each order
            total = Decimal('0')
            for _ in range(random.randint(1, 5)):
                dish = random.choice(Dish.objects.
                    filter(restaurant=restaurant))
                quantity = random.randint(1, 3)
                OrderItem.objects.create(
                    order=order,
                    dish=dish,
                    quantity=quantity,
                    unit_price=dish.price
                )
                total += dish.price * quantity

            order.total_amount = total
            order.save()
print("Fake data created successfully!")

```

and run it:

```
python createFakeData.py
```

OK now, we can see our data in the admin, and you will notice that our primary keys are now **UUID** instead of **integer**.

Chaining Filters

With all that in place, let’s now see how to create our chained filters. As shown on Figure 9-6, our goal is to display in the **Dish** list view a restaurant filter, and then based on the selected restaurant, we will only display the categories and dishes associated with that restaurant (see Figure 9-7 and Figure 9-8 for final result).

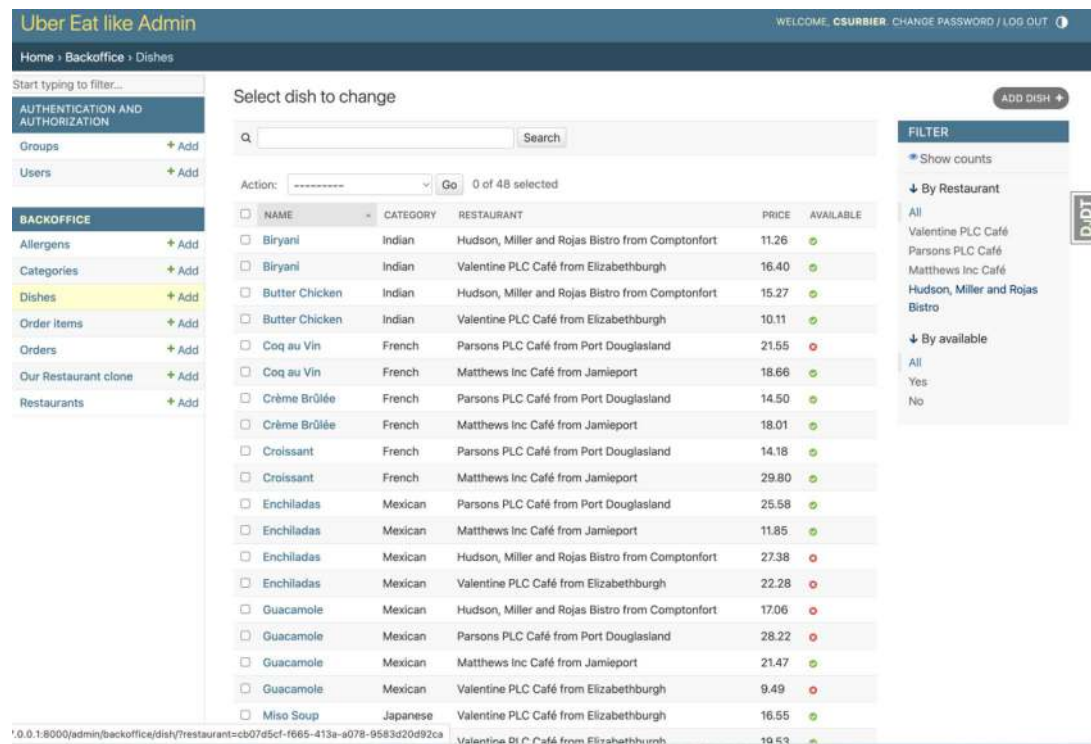


Figure 9-6. Dish changelist with a Restaurant filter (initial state with all restaurants)

Uber Eat like Admin

WELCOME, CSURBIER. CHANGE PASSWORD / LOG OUT

Home » Backoffice » Dishes

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

BACKOFFICE

Allergens + Add

Categories + Add

Dishes + Add

Order items + Add

Orders + Add

Our Restaurant clone + Add

Restaurants + Add

Select dish to change

Search 12 results (48 total)

Action: Go 0 of 12 selected

NAME	CATEGORY	RESTAURANT	PRICE	AVAILABLE
Coq au Vin	French	Parsons PLC Café from Port Douglasland	21.55	
Crème Brûlée	French	Parsons PLC Café from Port Douglasland	14.50	
Croissant	French	Parsons PLC Café from Port Douglasland	14.18	
Enchiladas	Mexican	Parsons PLC Café from Port Douglasland	25.58	
Guacamole	Mexican	Parsons PLC Café from Port Douglasland	28.22	
Pasta Carbonara	Italian	Parsons PLC Café from Port Douglasland	14.72	
Pizza Margherita	Italian	Parsons PLC Café from Port Douglasland	9.15	
Quesadillas	Mexican	Parsons PLC Café from Port Douglasland	11.17	
Ratatouille	French	Parsons PLC Café from Port Douglasland	18.99	
Risotto	Italian	Parsons PLC Café from Port Douglasland	17.60	
Tacos	Mexican	Parsons PLC Café from Port Douglasland	14.89	
Tiramisu	Italian	Parsons PLC Café from Port Douglasland	23.50	

12 dishes

FILTER

Show counts

Clear all filters

By Restaurant

All

Valentine PLC Café

Parsons PLC Café

Matthews Inc Café

Hudson, Miller and Rojas Bistro

By category

All

French

Italian

Mexican

By available

All

Yes

No

Figure 9-7. Chained filters: categories list updated after selecting a restaurant

Uber Eat like Admin

WELCOME, CSURBIER. CHANGE PASSWORD / LOG OUT

Home » Backoffice » Dishes

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

BACKOFFICE

Allergens + Add

Categories + Add

Dishes + Add

Order items + Add

Orders + Add

Our Restaurant clone + Add

Restaurants + Add

Select dish to change

Search 4 results (48 total)

Action: Go 0 of 4 selected

NAME	CATEGORY	RESTAURANT	PRICE	AVAILABLE
Enchiladas	Mexican	Parsons PLC Café from Port Douglasland	25.58	
Guacamole	Mexican	Parsons PLC Café from Port Douglasland	28.22	
Quesadillas	Mexican	Parsons PLC Café from Port Douglasland	11.17	
Tacos	Mexican	Parsons PLC Café from Port Douglasland	14.89	

4 dishes

FILTER

Show counts

Clear all filters

By Restaurant

All

Valentine PLC Café

Parsons PLC Café

Matthews Inc Café

Hudson, Miller and Rojas Bistro

By category

All

French

Italian

Mexican

By available

All

Yes

No

Figure 9-8. Chained filters: dishes filtered by the selected restaurant and category

Let's implement our RestaurantFilter as we learned in Chapter 2:

```
class RestaurantFilter(admin.SimpleListFilter):
    title = 'Restaurant'
    parameter_name = 'restaurant'

    def lookups(self, request, model_admin):
        # Get all restaurants for the filter dropdown options
        restaurants = Restaurant.objects.filter(is_active=True)
        return [(r.id, r.name) for r in restaurants]

    def queryset(self, request, queryset):
        # When a restaurant is selected, filter dishes
        if self.value():
            # Filter dishes directly by restaurant
            return queryset.filter(restaurant=self.value())
        return queryset
```

and then register it in our Dish model:

```
@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'restaurant', 'price', 'is_
    available',)
    list_filter = (RestaurantFilter, 'is_available',)
    list_select_related = ('category', 'restaurant',)
    search_fields = ('name', 'description')
    filter_horizontal = ('allergens',)

    def get_queryset(self, request):
        return super().get_queryset(request).prefetch_related('allergens')
        .select_related('category').select_related('restaurant')
```

OK and now we will implement our CategoryFilter:

```
class CategoryFilter(admin.SimpleListFilter):
    title = 'category'
    parameter_name = 'category'

    def lookups(self, request, model_admin):
```

```

# Get the selected restaurant ID from the request
restaurant_id = request.GET.get('restaurant')
if restaurant_id:
    # Get categories that have dishes in this restaurant
    # This ensures we only show relevant categories
    categories = Category.objects.filter(
        dishes_category__restaurant=restaurant_id
    ).distinct()
else:
    print("===No restaurant selected===")
    categories = Category.objects.all()

return [(c.id, str(c)) for c in categories]

def queryset(self, request, queryset):
    if self.value():
        return queryset.filter(category_id=self.value())
    return queryset

```

and then register it in our Dish model:

```

@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'price', 'is_available',)
    list_filter = (RestaurantFilter, CategoryFilter, 'is_available',)

```

That's it! Now you can filter dishes by restaurant and category, and the categories will only be displayed for the selected restaurant (if any). The magic happens in the `CategoryFilter` class, in the **lookups** method, where we check if a restaurant is selected and filter categories accordingly:

```

def lookups(self, request, model_admin):
    # Get the selected restaurant ID from the request (if any)
    restaurant_id = request.GET.get('restaurant')
    if restaurant_id:
        # Get categories that have dishes in this restaurant
        # This ensures we only show relevant categories
        categories = Category.objects.filter(

```

```

        dishes_category__restaurant=restaurant_id
    ).distinct()
else:
    print("===No restaurant selected===")
    categories = Category.objects.all()

    return [(c.id, str(c)) for c in categories]

```

To check if a restaurant is selected, we use the `request.GET.get('restaurant')` method. If a restaurant is selected, we filter dishes categories based on that restaurant. If no restaurant is selected, we display all categories.

list_display: Display a Database View

Another useful feature of the Django admin is the ability to display data which come from a database view and not from one of our Django models. It is super easy to do that.

Let's imagine our database team has created a view in our database called **ReportingStats** which contains some reporting information. If we want to display this view in the Django Admin, we just need to create a model in our **models.py** file with the fields that the table contains and tell Django what the database table name is and also that it is not managed by Django. To do that, we can use the **Meta** class and just need to add the **managed = False** and **db_table** attributes:

```

class ReportingStats(models.Model):
    month = models.DateField()
    nb_restaurants = models.IntegerField()
    nb_total_orders = models.IntegerField()
    class Meta:
        managed = False
        db_table = 'app_reportingstats'

```

Note The table name in the database must be the same as the one in the **Meta** class.

and then you can reference the model in the **admin.py** as usual:

```
@admin.register(ReportingStats)
class ReportingStatsAdmin(admin.ModelAdmin):
    list_display = ('month', 'nb_restaurants', 'nb_total_orders')
```

list_display: Proxying a Model

If we want the same model registered twice in the admin (two changelists/two different `ModelAdmin` setups for one table), we cannot register the same model class twice in `admin.py`. We need a proxy model in **models.py**, then register the proxy with a second `ModelAdmin` in **admin.py**:

```
class RestaurantClone(Restaurant):
    class Meta:
        proxy=True
        verbose_name_plural = "our Restaurant clone"
```

A proxy model is a subclass of an existing model whose `Meta` has `proxy = True`:

- It does not get its own database table.
- It uses the same table as the parent model (here, `Restaurant`).
- It exists so we can have a second model class for the same rows, with different default ordering, managers, and methods.

Our proxy is abstracting the `Restaurant` class. We can now register it in the admin with another `ModelAdmin` (see Figure 9-9):

```
@admin.register(RestaurantClone)
class RestaurantCloneAdmin(admin.ModelAdmin):
    list_display = ('name',)
    search_fields = ('name',)
```

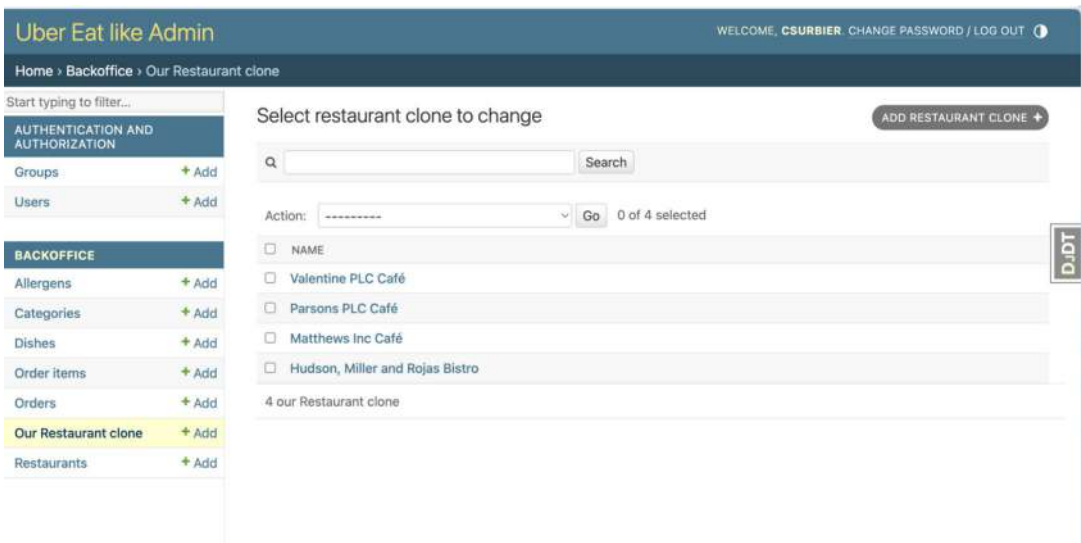


Figure 9-9. Proxy model registered in admin to provide a separate Restaurant “clone” entry

And now we can see our Restaurant clone in the admin and implement different **list_display**, **search_fields**, **list_filter**, ...

search_fields: custom get_search_results()

The **search_fields** attribute in a `ModelAdmin` class is great for basic search functionality, but sometimes we need more complex search features. Once again, Django allows us to completely customize the search behavior by overriding the `get_search_results()` method. In fact, as we will learn in this book, lots of default methods in the Django Admin are overridable, which lets us customize the behavior of the admin to our needs.

So with overriding the **get_search_results()** method, we can customize the search behavior to our needs. We can simply improve query performance, or we can implement a custom search algorithm. Let’s say we would like to implement a custom search keyword like **price:10** which will show us all restaurants which have dishes with a price greater than 10 (See Figure 9-10).

To do that, we can write a custom `get_search_results()` method in our `RestaurantAdmin` class:

```
def get_search_results(self, request, queryset, search_term):
    # Get the original queryset with default search
    queryset, use_distinct = super().get_search_results(request,
        queryset, search_term)

    # Custom logic: search with price range
    if search_term.startswith('price:'):
        try:
            # we get the value of the price
            price_value = float(search_term.split(':')[1])
            # we filter the restaurants with dishes with a price
            greater or equals than the price_value
            dish_with_price = Restaurant.objects.filter(
                dishes_restaurant__price__gte=price_value
            ).distinct()
            queryset = dish_with_price
        except Exception as e:
            print(e)

    return queryset, use_distinct
```

Within the method, we can get the value of what the user has searched for, by using the `search_term` variable. Then using this `search_term` variable, if it contains our `price:` keyword, we can get the price value by splitting the string using the `:` and then get the price value wanted to then filter the restaurants with dishes with a price greater or equal to that price value (as shown on Figure 9-11, Figure 9-12). Finally, we can return our filtered new queryset.

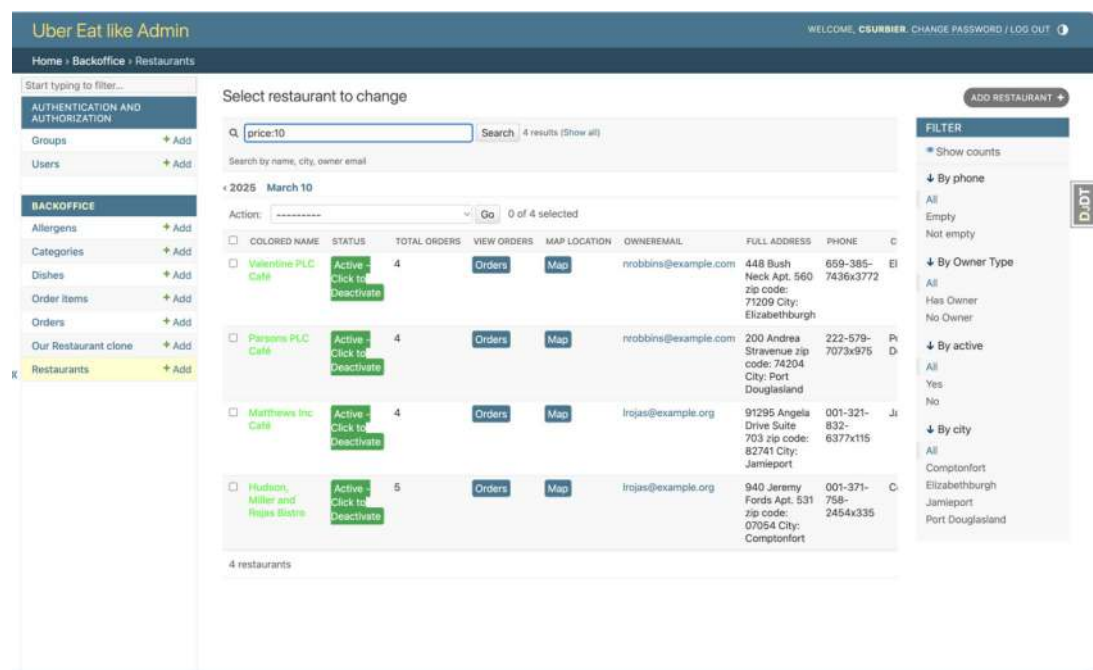


Figure 9-10. Custom search keyword `price:10` returning restaurants with dishes priced above the threshold

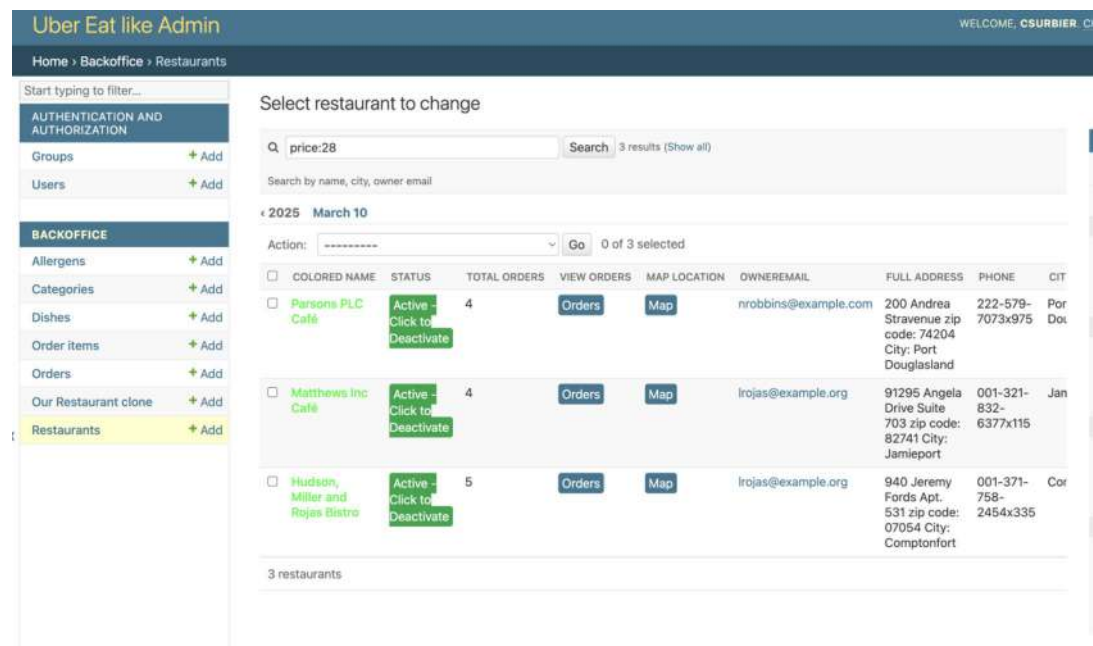


Figure 9-11. Custom search keyword `price:28` applied to filter results

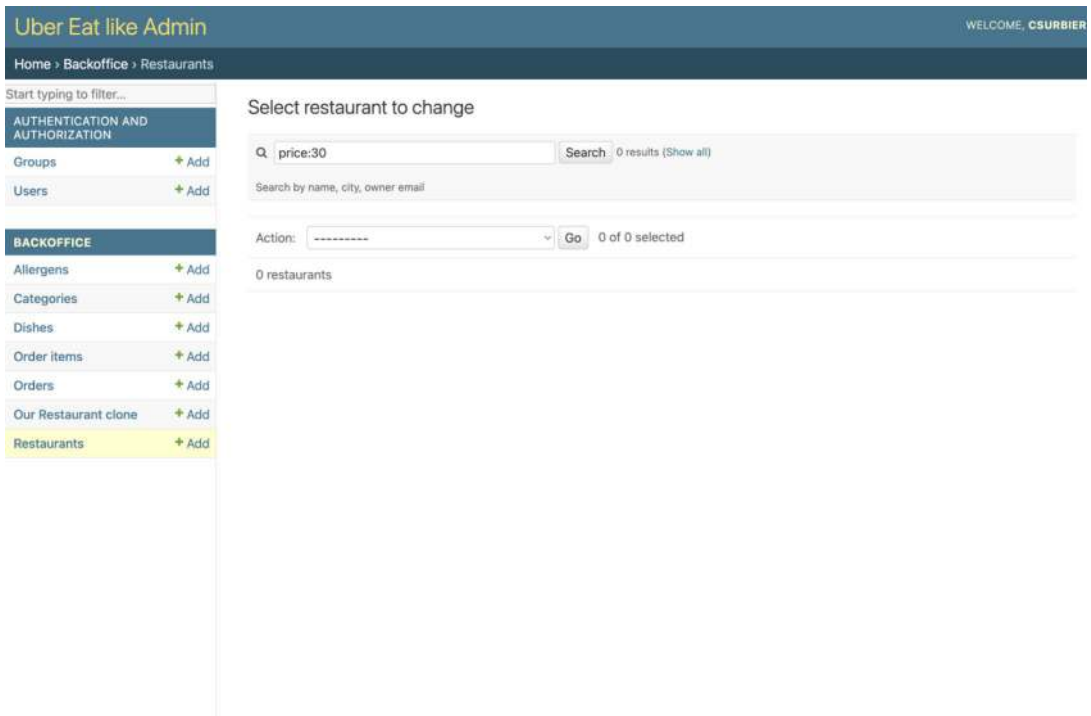


Figure 9-12. Custom search keyword `price:30` applied to filter results

And as we learned if we want to show the user, what he can search with, we just need to update the `search_help_text` attribute in the `RestaurantAdmin` class and add our custom `price:value` search keyword:

```
search_help_text = (_("Search by name, city, owner email,price:<value>"))
```

Note The `queryset` should always return a `Restaurant` object list, which is obvious because we are on the `Restaurant` model.

change_view: Display a change_form with Prefilled Values

We saw that **Django Admin** provides a default `change_form` view (see Figure 9-13) for each model that we can use to edit the data of a model instance. But what if we want to prefill the form with some default values?

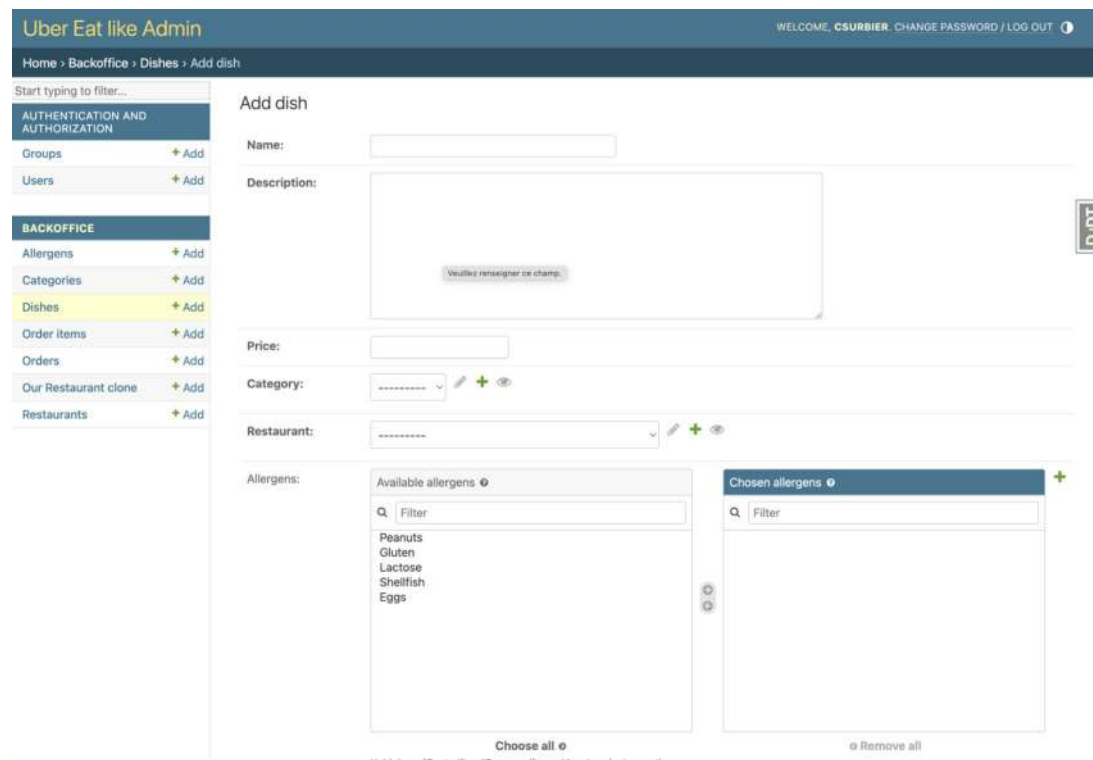


Figure 9-13. Dish change form opened with empty default values

When showing up, the form is empty.

Let's say that in 90% of the cases, we want to add a dish to a restaurant with the **French** category. It will be time-consuming to manually fill the category field every time we want to add a dish. So we can use the `change_view` attribute to prefill the form with our default Category values. We just need to override the `change_view` method in our **DishAdmin** class:

```
def get_form(self, request, obj=None, **kwargs):
    form = super(DishAdmin, self).get_form(request, obj=obj, **kwargs)
    category = Category.objects.filter(name="French").first()
    print("==Set with category==",category)
    form.base_fields['category'].initial = category
    return form
```

Now the form will be prefilled with the **French** category (Figure 9-14).

The screenshot shows a web application titled "Uber Eat like Admin". The top navigation bar includes a welcome message for "CSURBIER" and links for "CHANGE PASSWORD" and "LOG OUT". The breadcrumb trail indicates the user is in "Home > Backoffice > Dishes > Add dish".

The left sidebar contains a menu with sections: "AUTHENTICATION AND AUTHORIZATION" (Groups, Users), "BACKOFFICE" (Allergens, Categories, Dishes, Order items, Orders, Our Restaurant clone, Restaurants), and "RESTAURANTS". The "Dishes" option is highlighted.

The main content area is titled "Add dish". It contains the following fields:

- Name:** An empty text input field.
- Description:** A large text area.
- Price:** A text input field.
- Category:** A dropdown menu showing "French" as the selected option.
- Restaurant:** A dropdown menu with a placeholder "-----".
- Allergens:** A section with two panels:
 - Available allergens:** A list containing "Peanuts", "Gluten", "Lactose", "Shellfish", and "Eggs".
 - Chosen allergens:** An empty list.

Figure 9-14. Dish change form prefilled with a default Category using `get_form()`

To add a default price (Figure 9-15), we could have set the default value of the **price** field in our **model** class. But we can also do it in the **get_form** method:

```
def get_form(self, request, obj=None, **kwargs):
    form = super(DishAdmin, self).get_form(request, obj=obj, **kwargs)
    category = Category.objects.filter(name="French").first()
    print("==Set with category==",category)
```

```

form.base_fields['category'].initial = category
form.base_fields['price'].initial = 10
return form

```

The screenshot shows the 'Add dish' form in the 'Uber Eat like Admin' interface. The form is pre-filled with a default Price of 10. The interface includes a sidebar with navigation links and a main content area with form fields for Name, Description, Price, Category, Restaurant, and Allergens.

Uber Eat like Admin WELCOME, CSURBIER CHANGE PASSWORD / LOG OUT

Home > Backoffice > Dishes > Add dish

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

BACKOFFICE

Allergens + Add

Categories + Add

Dishes + Add

Order Items + Add

Orders + Add

Our Restaurant clone + Add

Restaurants + Add

Add dish

Name:

Description:

Price:

Category:

Restaurant:

Allergens:

Available allergens

Filter

Peanuts

Gluten

Lactose

Shellfish

Eggs

Choose all

Chosen allergens

Filter

Remove all

Figure 9-15. Dish change form prefilled with a default Price using `get_form()`

We just need to set the **initial** attribute of the field we want to prefill:

```
form.base_fields['category'].initial = category
```

or

```
form.base_fields['price'].initial = 10
```

change_view: Forms Widget

Django comes in the box with a lot of form widgets that we can use to display our data. For example, let's say that we want to display a **textarea** widget (see Figure 9-16) for the **address** field. We can do that by adding the **widgets** attribute to our **RestaurantForm** class:

```
class RestaurantForm(forms.ModelForm):
    confirm_phone = forms.CharField(max_length=20,required=False)
    address = forms.CharField(
        widget=forms.Textarea(attrs={
            'rows': 4,
            'class': 'special-textarea'
        })
    )
class Meta:
    model = Restaurant
    fields = '__all__'
```

The screenshot shows the Django Admin interface for 'Uber Eat like Admin'. The top navigation bar includes 'Home', 'Backoffice', 'Restaurants', and 'Add restaurant'. The left sidebar contains a menu with 'AUTHENTICATION AND AUTHORIZATION' (Groups, Users) and 'BACKOFFICE' (Allergens, Categories, Dishes, Order items, Orders, Our Restaurant clone, Restaurants). The main content area is titled 'Add restaurant' and contains a form with the following fields: 'Name' (text input), 'Image' (file upload), 'Address' (textarea with a custom class 'special-textarea' and 4 rows), 'Zip code' (text input), 'City' (text input), 'Country' (text input), 'Phone' (text input), 'Confirm phone' (text input), and 'Opening hours' (time picker). A message 'Veuillez renseigner ce champ.' is displayed below the address field.

Figure 9-16. Custom form widget (textarea) rendered in the admin change form

Let's imagine we added an **opening_hours** field to our **Restaurant** model. We can use the **TimeField** widget to display a time picker:

```

class RestaurantForm(forms.ModelForm):
    confirm_phone = forms.CharField(max_length=20,required=False)
    address = forms.CharField(
        widget=forms.Textarea(attrs={
            'rows': 4,
            'class': 'special-textarea'
        })
    )

    opening_hours = forms.TimeField(
        widget=forms.TimeInput(attrs={
            'type': 'time'
        })
    )

class Meta:
    model = Restaurant
    fields = '__all__'

```

This is just for demo purposes since we don't have an **opening_hours** field in our **Restaurant** model. Just add it to our **fields** list.

```

fieldsets = (
    ("My first section", {
        'fields': ('name', 'opening_hours','image', ('address',
        'zip_code', 'city'), 'country', 'phone','confirm_phone', ),
        'description': 'This is the first section of the form.'
    }),
    ('Advanced options', {
        'classes': ('collapse',),
        'fields': ('is_active','owner',),
    }),
)

```

List of all available widgets can be found [in the documentation \(https://docs.djangoproject.com/en/5.1/ref/forms/widgets/\)](https://docs.djangoproject.com/en/5.1/ref/forms/widgets/).

change_view: Display a change_form with Foreign Key Restriction

Now let's say that we want to restrict the user to add a dish only to a restaurant that contains the word "Val" in its name (see Figure 9-17). We can use the `formfield_for_foreignkey` method to do that.

```
@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):

    def formfield_for_foreignkey(self, db_field, request, **kwargs):
        if db_field.name == "restaurant":
            kwargs["queryset"] = Restaurant.objects.filter(name__icontains="Val")
        return super().formfield_for_foreignkey(db_field, request, **kwargs)
```

The screenshot shows the 'Add dish' form in the 'Uber Eat like Admin' interface. The form includes the following fields:

- Name:** A text input field.
- Description:** A large text area.
- Price:** A text input field with the value '10'.
- Category:** A dropdown menu showing 'French'.
- Restaurant:** A dropdown menu showing 'Valentine PLC Café from Elizabethburgh'.
- Allergens:** A section with two panels: 'Available allergens' and 'Chosen allergens'. The 'Available allergens' panel lists: Peanuts, Gluten, Lactose, Shellfish, and Eggs.

Figure 9-17. ForeignKey choices restricted in the form using `formfield_for_foreignkey()`

Note For many-to-many fields, we can use the `formfield_for_manytomany` method. For choice fields, we can use the `formfield_for_choice_field` method.

change_view: Initialize an Inline Formset and Do Some Custom Validation

If we configure an inline in an admin class, we can also initialize it. Let's add an inline to our **RestaurantAdmin** class to be able to add dishes to the restaurant at the same time:

```
class DishInline(admin.StackedInline):
    model = Dish
    extra = 1
    show_change_link = True

    and

@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    inlines = [DishInline]
```

Now, when adding a restaurant, we can add dishes to it, but fields will be empty (Figure 9-18).

Figure 9-18. Restaurant change form with an inline Dish formset (blank inline row)

We will do the same as before, setting default values for the **category** and **price** fields. The supported way to do that is to override **formfield_for_dbfield** on **DishInline**.

Django calls **formfield_for_dbfield** for each model field as it builds the inline's forms:

```
class DishInline(admin.StackedInline):
    model = Dish
    extra = 1
    show_change_link = True

    def formfield_for_dbfield(self, db_field, request, **kwargs):
        if db_field.name == "price":
            kwargs["initial"] = 10
        if db_field.name == "category":
```

```

    if not hasattr(self, "_french_category_cache"):
        self._french_category_cache = Category.objects.filter(
            name="French"
        ).first()
    kwargs["initial"] = self._french_category_cache
return super().formfield_for_dbfield(db_field, request, **kwargs)

```

1. **formfield_for_dbfield** is the hook the admin uses to customize each `FormField` (labels, widgets, and **initial** data) for a given `db_field`.
2. We set **initial** for **price** to 10 and for **category** to the first **Category** named "French". The `hasattr / _french_category_cache` pattern runs that lookup **once** per inline, even if Django builds more than one empty form when **extra is greater than 1**. It avoids multiple queries to the database.
3. We end with **super()** so default behavior (widgets, `ModelChoice` queryset, validation) is preserved.

And now we have our inline formset filled with default values (Figure 9-19).

The screenshot shows a web application titled "Uber Eat like Admin". The top navigation bar includes "Home", "Backoffice", "Restaurants", and "Add restaurant". A sidebar on the left lists various admin functions like "Groups", "Users", "Allergens", "Categories", "Dishes", "Order items", "Orders", "Our Restaurant clone", and "Restaurants". The main content area is titled "Add restaurant" and contains a form with several sections. The "My first section" includes fields for Name, Image (with a file upload button), Address, City, Country, Phone, and Confirm phone. Below this is an "Advanced options" section. The "DISHES" section is currently active, showing a "Dish: #1" form. This form has fields for Name, Description, Price (set to 10), Category (set to French), and Allergens (Peanuts, Gluten).

Figure 9-19. *Inline Dish formset initialized with default category and price values*

Now let's add some validation to our inline formset. We want to validate that the **price** field is greater than 5 (for whatever reason). We can create our own form validation class by overriding the **clean** method of the **BaseInlineFormSet** class:

```
class DishInlineFormSet(forms.BaseInlineFormSet):
```

```
    def clean(self):
        super(DishInlineFormSet, self).clean()
```

and then write our validation logic:

```
class DishInlineFormSet(forms.BaseInlineFormSet):
```

```
    def clean(self):
        super().clean()
        # example custom validation across forms in the formset
        for form in self.forms:
```

```
# your custom formset validation
price = form.cleaned_data.get('price')
if price and price <= 5:
    raise forms.ValidationError("Price must be greater than 5")
```

Now if we try to add a dish with a price lower than 5 and try to save the restaurant, we will have an error message (Figure 9-20).

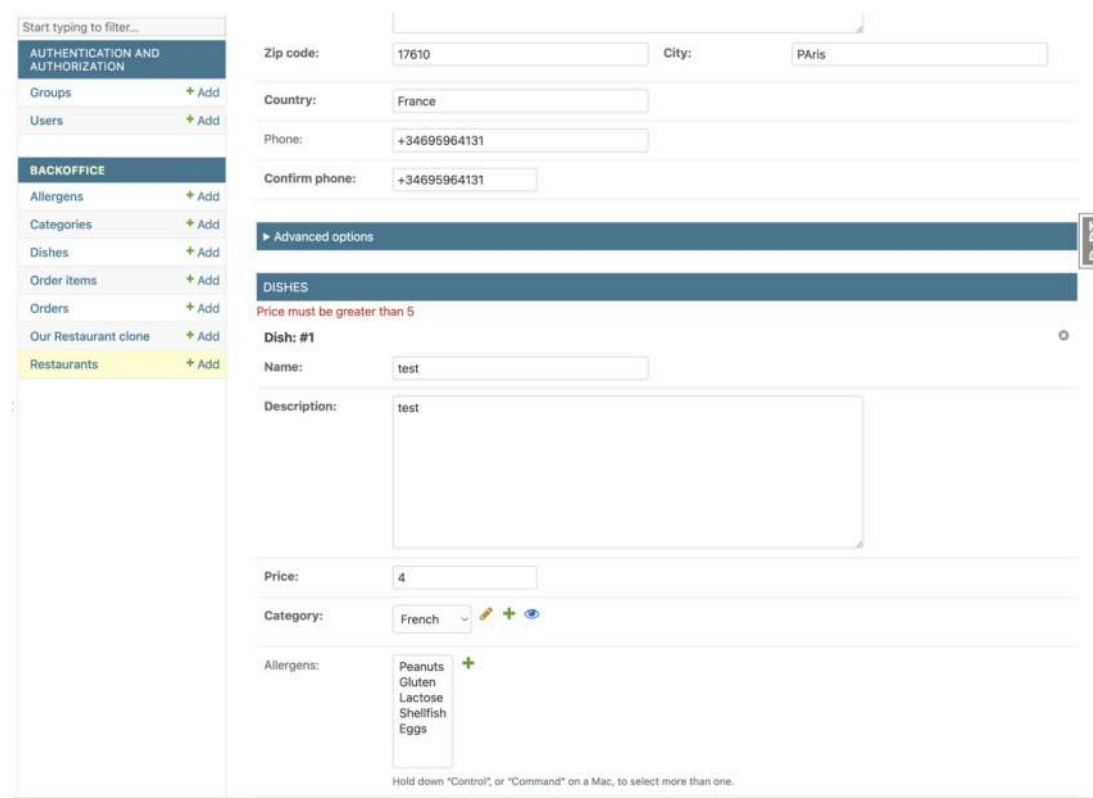


Figure 9-20. *Inline formset validation error displayed when saving (price must be >5)*

change_view: raw_id_fields

Let’s say our **Category** dataset is huge; for performance reasons, we will use the `raw_id_` field option in our inline which will show us a search box instead of a select field:

```
class DishInline(admin.StackedInline):
    formset = DishInlineFormSet
```

```

model = Dish
raw_id_fields = ["category"]
extra = 1
show_change_link = True

```

And now the interface will look as shown in Figure 9-21.

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#)

Users [+ Add](#)

BACKOFFICE

Allergens [+ Add](#)

Categories [+ Add](#)

Dishes [+ Add](#)

Order items [+ Add](#)

Orders [+ Add](#)

Our Restaurant clone [+ Add](#)

Restaurants [+ Add](#)

Advanced options

DISHES

Dish: #1

Name:

Description:

Price:

Category: [Q French](#) [Lookup](#)

Allergens: [+](#)

Hold down "Control", or "Command" on a Mac, to select more than one.

☒ Available

Image: Aucun fichier choisi

[+ Add another Dish](#)

[SAVE](#) [Save and add another](#) [Save and continue editing](#)

7.0.0.1:8000/admin/backoffice/category/?_to_field=id

Figure 9-21. *Inline ForeignKey field rendered as raw_id_fields for performance*

And when clicking on the search icon, a pop-up will open with the **Category** search interface (Figure 9-22).

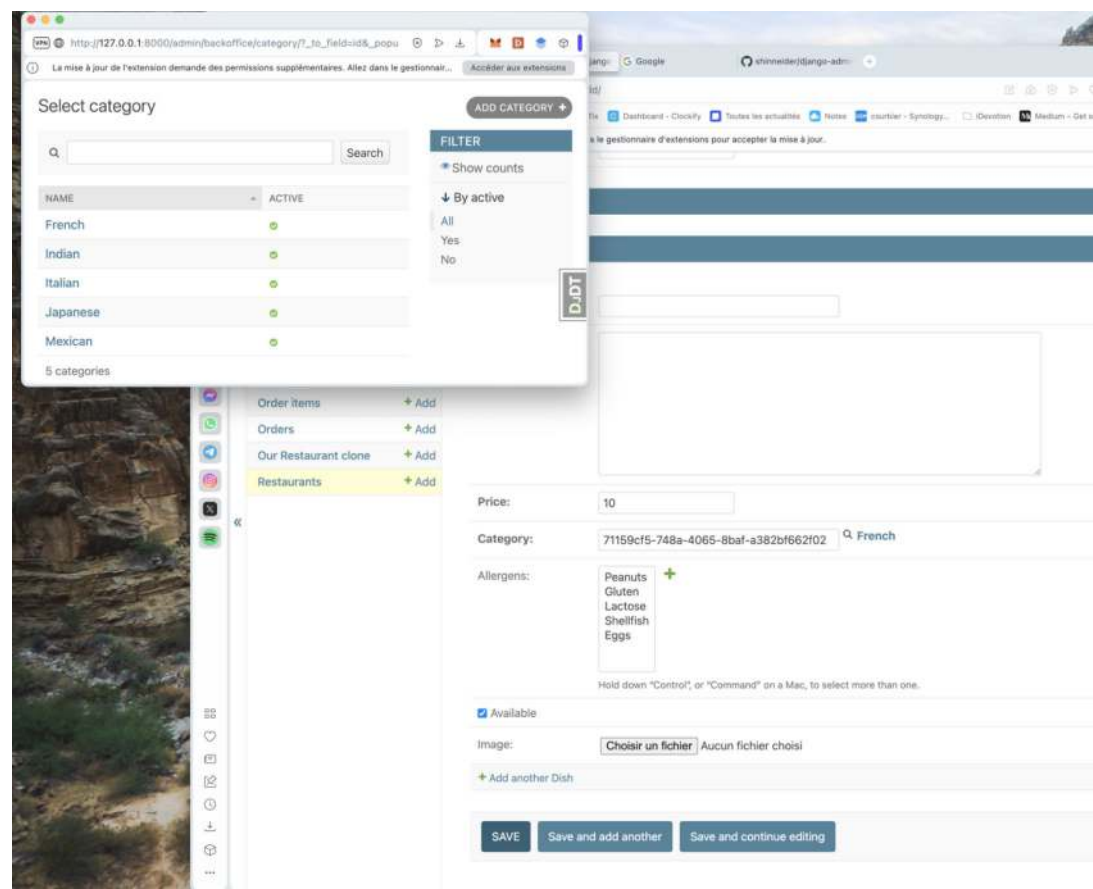


Figure 9-22. Pop-up search window opened from a `raw_id_fields` lookup icon

change_view: Inline Pagination

When editing a restaurant, our DishInline will be displayed with all the existing dishes for that restaurant, so it can be a lot of dishes. For the same performance reasons, we need to limit the resultset and use pagination in our inline.

Django Admin doesn't provide a way to do that out of the box, but we can use a third-party package to do that.

First, we need to install the **django-admin-inline-paginator** (<https://github.com/shinneider/django-admin-inline-paginator>) package:

```
pip install django-admin-inline-paginator
```

Then we need to add it to our **INSTALLED_APPS** list in our **settings.py** file:

```
INSTALLED_APPS = [
    ...
    'django_admin_inline_paginator',
]
```

And finally, we can use it in our **DishInline** class:

```
from django_admin_inline_paginator.admin import TabularInlinePaginated
...
class DishInline(TabularInlinePaginated):
    formset = DishInlineFormSet
    model = Dish
    raw_id_fields = ["category"]
    extra = 1
    show_change_link = True
    per_page = 2
```

1. We need to import the **TabularInlinePaginated** class from the **django_admin_inline_paginator** package.
2. We need to set the **per_page** attribute to the number of items we want to display per page.
3. We need to set the **TabularInlinePaginated** class as our **DishInline** class.

And now we have our inline formset with pagination (Figure 9-23).

The screenshot shows a Django admin form for a 'Restaurant' model. On the left is a sidebar with a search bar and a list of categories: 'Authentication and Authorization', 'Office', 'Dishes', 'Menu', 'Restaurant clone', and 'Dishes'. The main form area has fields for 'Phone' (222-579-7073x975) and 'Confirm phone'. Below these is an 'Advanced options' section. The 'DISHES' section is an inline formset with a table-like structure. It has columns for 'NAME', 'DESCRIPTION', and 'PRICE'. The first dish is 'Coq au Vin' with a price of 21,55. The second dish is 'Crème Brûlée' with a price of 14,50. There is a '10' in the price column for a third, empty dish. At the bottom of the formset is a button 'Add another Dish'. Below the formset is a pagination bar showing '1 2 3 4 5 6 next 12 Results'. At the very bottom are buttons for 'SAVE', 'Save and add another', 'Save and continue editing', and 'Delete'.

Figure 9-23. *Paginated inline formset using django-admin-inline-paginator*

change_view: Some Restrictions

One useful feature when creating a **new restaurant** with a new inline formset could be to check some parameters in the inline formset (our Dishes) before saving the restaurant. Let’s imagine that we want to check if the restaurant has at least one dish created and if so set the **is_active** field to **True** (or False).

The challenge is accessing the inline formset data from our **Restaurant** main form’s validation method. In Django, this is not directly possible because the form doesn’t have access to the formsets. So in our **RestaurantForm** class, in our **clean** method, we can’t access the **DishInlineFormSet** formset, so we can’t do some extra custom logic validation.

Add Actions with Intermediate Page

In Chapter 2, we have seen how to add actions to our Django Admin interface. And we have seen that the action is executed directly on the queryset.

But we can also add actions with a custom page (see Figure 9-24). To do so, instead of executing our queryset, we can return a view with a custom template to which we are passing (in the context) the selected items on the admin interface.



Figure 9-24. *Intermediate confirmation page for a custom admin action*

And then when clicking the **Make inactive** button, we will have the result shown in Figure 9-25.

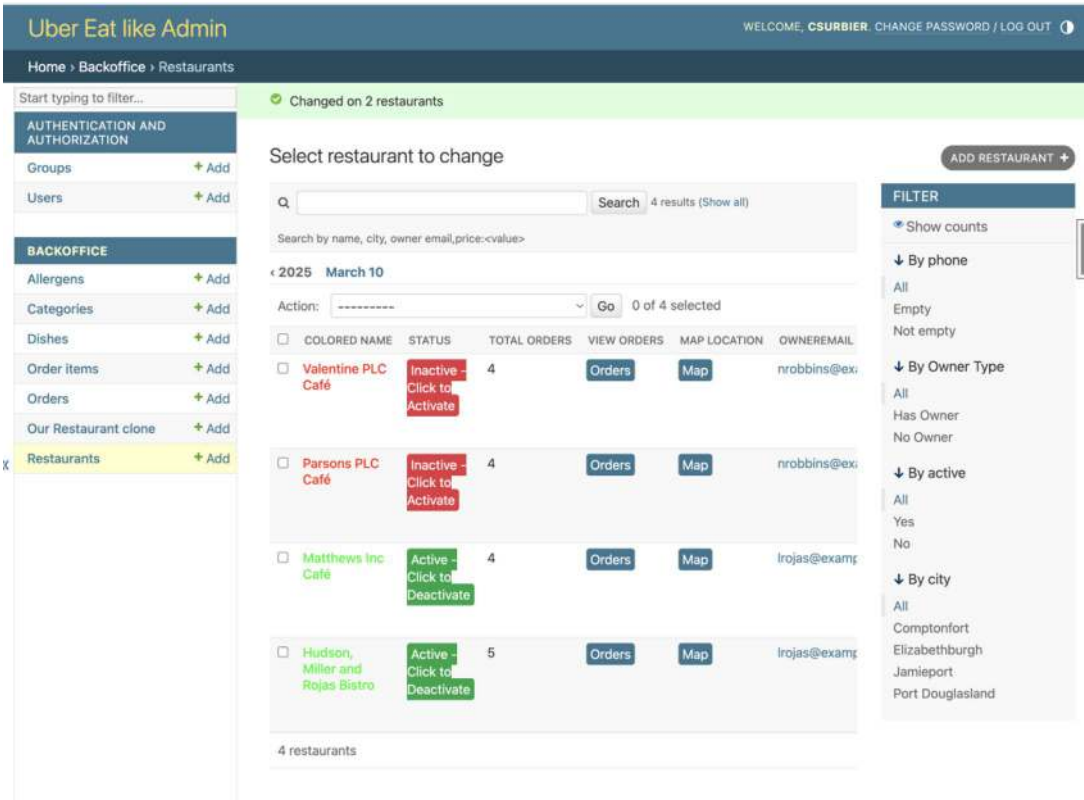


Figure 9-25. Result after applying the custom action from the intermediate page

OK, let’s modify our existing method **make_inactive**:

```
@admin.action(
    description="Mark selected restaurants as inactive"
)
def make_inactive(self, request, queryset):
    updated = queryset.update(is_active=False)
    self.message_user(
        request,
        f"{updated} restaurants were successfully marked as inactive."
    )
```

```

to return a view with a custom template

from django.shortcuts import render
...

@admin.action(
    description="Mark selected restaurants as inactive with
custom view"
)
def make_inactive(self, request, queryset):
    return render(request,
        'admin/backoffice/custom_make_inactive.html',
        context={'restaurants':queryset}
    )

```

When the method will be executed, it will return a view with a custom template, and the queryset with our **selected restaurants** will be passed as a context variable. This is the **render** function that will be used to render our template.

So at the root directory of our project, let's create a **templates** directory. Then create subdirectories **admin/backoffice** and then create a file **custom_make_inactive.html** file (check Figure 9-26).

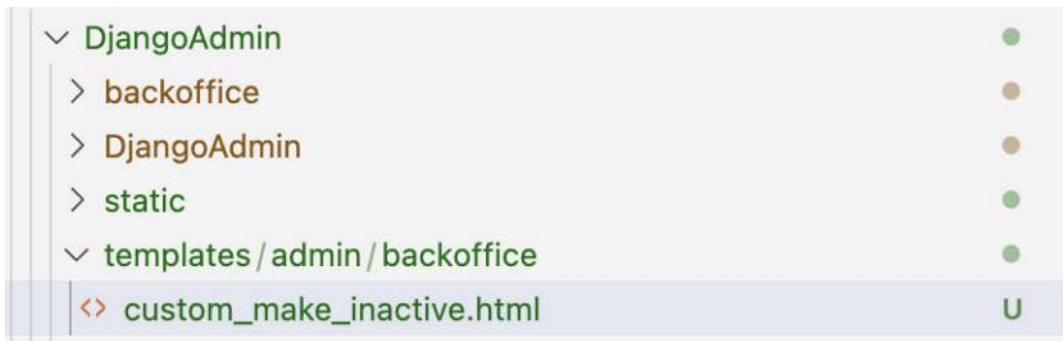


Figure 9-26. Project template directory structure for overriding admin templates

We need to modify our **settings.py** file to update the **TEMPLATES** variable and add our current project directory using the **BASE_DIR** variable:

```

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',

```

```

'DIRS': [
    os.path.join(BASE_DIR, 'templates'), # Project-level templates
    directory
],
'APP_DIRS': True,
'OPTIONS': {
    'context_processors': [
        'django.template.context_processors.debug',
        'django.template.context_processors.request',
        'django.contrib.auth.context_processors.auth',
        'django.contrib.messages.context_processors.messages',
    ],
},
},
]

```

Note The ‘DIRS’: [] setting defines a list of directories where Django will look for template files. When Django needs to render a template, it searches through these directories in the order they’re listed.

```

'DIRS': [
    os.path.join(BASE_DIR, 'templates'), # Project-level templates
    directory
],

```

This would tell Django to look in a ‘templates’ directory at the root of our project before checking app-level template directories.

The ‘APP_DIRS’: True setting tells Django to look for templates in the “templates” subdirectory of each installed application. In our case, it will look for a templates directory in our **backoffice** app.

Now we will write our HTML template file. We will learn more about Django templates in the next section, but just know for now that we can inherit from default Django templates and override some of the content.

```
{% extends "admin/base_site.html" %}
{% block extrahead %}
{{ block.super }}
```

<style>

```
.restaurant-name {
    color: red;
    font-weight: bold;
}
```

</style>

```
{% endblock %}
```

```
{% block content %}
```

Are you sure you want to make inactive these restaurants ?

```
{% endblock %}
```

Then we need to display the default form (provided by django) for our selected restaurants. This form needs three important properties:

1. An action key which should map to our admin action. We called it **make_inactive** in our admin action.
2. An apply key which we will use in the admin to determine we are applying the action.
3. The selected items, in the form of `_selected_action`.

```
{% extends "admin/base_site.html" %}
{% block extrahead %}
{{ block.super }}
```

<style>

```
.restaurant-name {
    color: red;
    font-weight: bold;
}
```

</style>

```
{% endblock %}

{% block content %}
<form action="" method="post">
    {% csrf_token %}
    <p>
        Are you sure you want to make inactive these restaurants ?
    </p>
    {% for restaurant in restaurants %}
        <p>
            <span class="restaurant-name">{{ restaurant }}</span>
        </p>
        <input type="hidden" name="_selected_action" value="{{ restaurant.
            pk }}" />
    {% endfor %}

    <input type="hidden" name="action" value="make_inactive" />
    <input type="submit" name="apply" value="Make Inactive" />
</form>
{% endblock %}
```

Now we can modify our **make_inactive** method to handle the POST request:

```
@admin.action(
    description="Mark selected restaurants as inactive with
    custom view"
)
def make_inactive(self, request, queryset):
    if 'apply' in request.POST:
        # The user clicked submit on the intermediate form.
        # Perform our update action:
        queryset.update(is_active=False)
        # Redirect to our admin view after our update has
        # completed with a nice little info message saying
        # our models have been updated:
        self.message_user(request,
            "Changed    on {} restaurants".format(queryset.count()))
```

```
return HttpResponseRedirect(request.get_full_path())  
  
return render(request,  
    'templates/admin/backoffice/custom_make_inactive.html',  
    context={'restaurants':queryset})
```

1. We check if the **apply** keyword is in the POST.
2. If so, we perform our update action and redirect to our admin view.
3. If not, we return our template.

We just saw how to use the Django template system. We will now dive a little bit into the template system.

CHAPTER 10

Template System Inheritance

Django’s template inheritance is one of its most powerful features, allowing us to build a base “skeleton” template that contains all the common elements of our admin site.

We can override many of the default Django Admin templates which the admin module uses to generate the various pages of the admin interface. We can even override a few of these templates for a specific app **OR** a specific model. This is done by creating a directory in our app’s templates directory with the same name as the template we want to override.

For example, to override the *change_form.html* template for our Backoffice app, we would create within our **backoffice** directory a **templates/admin/backoffice/** with a new **change_form.html** file. Then our Django Admin will use this template for all of our models instead of the default one.

If we want to override a template for a specific model such as our **Restaurant**, we can create a subdirectory with the same model name (in lowercase) that we want to override: **templates/admin/backoffice/restaurant/change_form.html**.

*Not every Django Admin template can be overridden per app or per model. The default Django Admin files are located in **django/contrib/admin/templates/admin**.*

```
ls ../venv/lib/python3.12/site-packages/django/contrib/admin/
templates/admin
404.html
base.html
change_list_results.html
filter.html
object_history.html
500.html
base_site.html
```


color_theme_toggle.html
pagination.html
actions.html
change_form.html
date_hierarchy.html
index.html
popup_response.html
app_index.html
change_form_object_tools.html
delete_confirmation.html
invalid_setup.html
prepopulated_fields_js.html
app_list.html
change_list.html
delete_selected_confirmation.html
login.html
search_form.html
nav_sidebar.html
submit_line.html

Table 10-1 is a comprehensive table of the main Django Admin template files and their purposes.

Table 10-1. *Main Django Admin template files and their purposes*

Template File	Purpose
actions.html	On the changelist (the table of all rows for a model), this block renders the action dropdown —built-in and custom <code>ModelAdmin</code> actions (e.g., “Delete selected ...”) that run on the rows you tick. This is <i>not</i> the same as the top object tools strip (e.g., “Add”); <i>actions</i> are batch operations on many rows.
app_index.html	Shown when you navigate into one app from the main admin home (e.g., <code>/admin/app_label/</code>). It lists that app’s registered models with links to each model’s changelist so you pick which model to work on.

(continued)

Table 10-1. *(continued)*

Template File	Purpose
<code>change_form.html</code>	The main layout for adding or changing a single object (URLs with <code>add/</code> or <code>.../change/</code>). This page is the change form (or “add/change” form): one record’s fieldsets, errors, inlines, and it usually includes the submit bar and the per-object object tools. It is <i>not</i> the list of all instances.
<code>change_form_object_tools.html</code>	Renders object tools on the <i>change form</i> : the small set of top links for this instance , e.g., History (if logging is on) and View on site (if the model provides it). <i>Object tools</i> = those per-row/per-object links, distinct from the actions dropdown and from list-level “Add”.
<code>change_list.html</code>	The main shell for the changelist : the “all objects” view for a model (search, filters, action dropdown, results table, pagination). Opening a model from the app index shows this.
<code>change_list_object_tools.html</code>	Renders object tools for the <i>changelist</i> —classically “ Add [Model name] ” and any similar list-scoped links. “For the model as a whole” means: they are about creating a new instance or other operations not tied to a checked row (contrast the change form’s object tools, which refer to one saved object).
<code>change_list_results.html</code>	The partial that prints only the result grid : column headers, row checkboxes, cell values, and the “no results” state. The surrounding changelist (search, filters, actions) is handled by other templates.
<code>date_hierarchy.html</code>	If <code>ModelAdmin</code> sets date_hierarchy to a <code>DateField</code> or <code>DateTimeField</code> , the admin can show a strip of drill-down links (by year, month, day) to narrow the changelist. This template builds those links. They are separate from, but work alongside, the list_filter sidebar.
<code>delete_confirmation.html</code>	Shown when you delete from the change form for one object. It lists the object, warns about cascading deletes, and requires confirmation before the <code>DELETE</code> runs.
<code>object_history.html</code>	Renders the change history page for a single object (from History in object tools), using <code>LogEntry</code> when <code>django.contrib.admin</code> logging is enabled: who did what, when.

(continued)

Table 10-1. *(continued)*

Template File	Purpose
pagination.html	Renders the paging controls (first, previous, page numbers, next, last) at the bottom of a long changelist, respecting <code>list_per_page</code> .
popup_response.html	For Add on a ForeignKey (or similar), the admin can open a child window to create a related row. The successful post returns a small HTML page in that window: JavaScript passes the new object's id and display text back to the parent form and closes the pop-up. This template is that popup response body—not a full admin page.
prepopulated_fields_js.html	Includes the script behind prepopulated_fields : e.g., copying a title into a slug as you type. Only relevant when the change form has prepopulated fields.
search_form.html	Renders the changelist search field and “Search” for <code>search_fields</code> on the <code>ModelAdmin</code> (different from the filter list in the sidebar, which is another mechanism).
submit_line.html	Renders the Save/Save and add another/Save and continue editing/Delete (on change)/sometimes View on site row at the bottom of the change form —the main buttons for the current form submission.

In part 4, we will study more in detail how to create our own custom admin template or use existing Django Admin packages to rebrand or extend the Django Admin.

For now, let's focus on customizing some of the Django Admin templates, in a step-by-step manner.

Change Login Form

1. Overriding the default admin file

As said previously, to override a default admin file, we need to create a new directory in the **templates** directory of our application Backoffice and create a new directory in the **templates/admin** directory and a new directory in the **templates/admin/backoffice** directory. Let's do this:

```
mkdir -p backoffice/templates/admin/backoffice
```

Then we need to copy the default login template from the **django/contrib/admin/templates/admin** directory to the **backoffice/templates/admin/** directory, because the **login.html** file is specific to the admin app (and not to our backoffice app):

```
cp ../../venv/lib/python3.12/site-packages/django/contrib/admin/templates/admin/login.html templates/admin/
```

Your project directory should now look as shown in Figure 10-1.

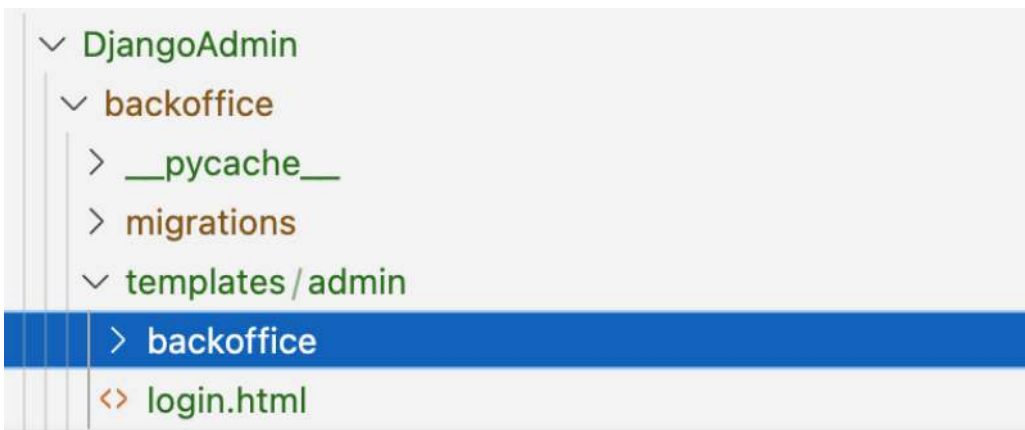


Figure 10-1. Admin template override files added to the project directory

The login template is just a form with username and password fields and a button to submit the form (Figure 10-2).

Figure 10-2. *Default Django Admin login form template content*

Let's customize the login form by adding a new label to the form. We need to edit the **login.html** file we just copied from the **django/contrib/admin/templates/admin** directory to the **backoffice/templates/admin/** directory:

...

```
<form action="{% app_path %}" method="post" id="login-
form">{% csrf_token %}
  <h4>My new Login FORM</h4>    <!-- OUR MODIFICATION -->
  <div class="form-row">
    {{ form.username.errors }}
    {{ form.username.label_tag }} {{ form.username }}
```

```

</div>
<div class="form-row">
    {{ form.password.errors }}
    {{ form.password.label_tag }} {{ form.password }}
    <input type="hidden" name="next" value="{{ next }}">
</div>
{% url 'admin_password_reset' as password_reset_url %}
{% if password_reset_url %}
<div class="password-reset-link">
    <a href="{{ password_reset_url }}">{% translate
    'Forgotten your password or username?' %}</a>
</div>
{% endif %}
<div class="submit-row">
    <input type="submit" value="{% translate 'Log in' %}">
</div>
</form>
...

```

I just added the `<h4>My new Login FORM</h4>` line to the login form.

Let's run the server and go to the login page. Nothing has changed! It's because Django searches for templates in the order **apps are listed**. So we need our 'backoffice' to come before 'django.contrib.admin' in the **INSTALLED_APPS** setting:

```

INSTALLED_APPS = [
    'backoffice',
    'django.contrib.admin',
    ...

```

And now we can refresh the page and see our custom login form (Figure 10-3).

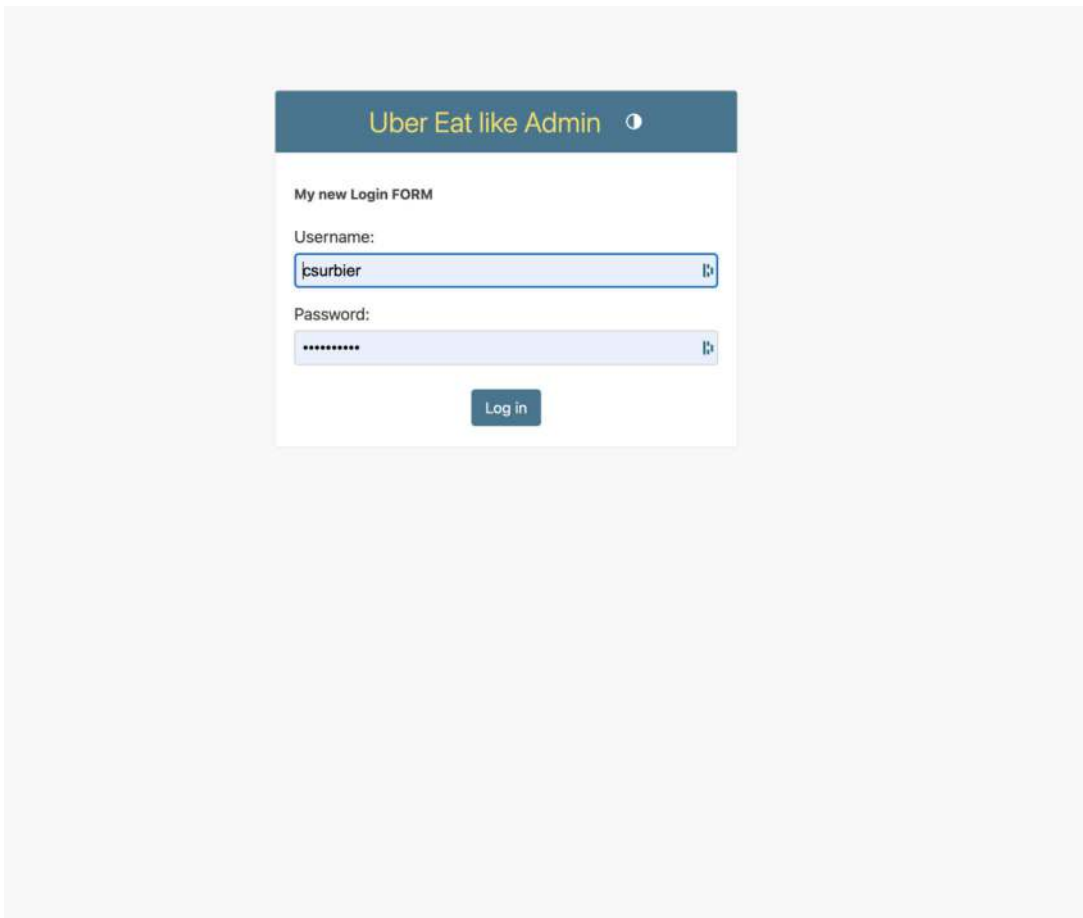


Figure 10-3. Custom label rendered after overriding the admin login template and app order

Wow, it's quite easy! We just need to be careful when overriding default admin templates and not delete important pieces Django relies on (template tags, `{% block %}` structure, forms, or scripts).

What we can change safely (most of the time): We can adjust **presentation**: add wrappers, classes, and ids for CSS; change layout and branding; add content inside blocks that are meant to be extended. We can load extra static CSS or JavaScript from **`{% block extrahead %}`/**`{% block extrastyle %}`** (or your base template's equivalents) so we **theme** the admin without touching**

every line of the stock file. When a block already exists, prefer **{{ block.super }}**, so we keep the default output and only add around it. Copy the template from the **same Django version** we run, then edit; that is safer than writing a new file from memory.

What we must not remove or break (without re-implementing behavior): Keep the **{% extends ... %}** chain and the **block names** the parent expects. Do not drop **{% load %}** lines for tags the template uses (`i18n`, `static`, `admin_urls`, etc.). Preserve **loops and {% if %}** that render rows, filters, actions, and object tools: they receive **context** from the view (`cl`, `opts`, `permissions`, etc.), and removing them removes features. Keep **forms** intact: **{% csrf_token %}**, `method="post"` where the original has them and field names the admin's JavaScript expects. Do not strip **<script>** / **<link>** includes that power inlines, related-object popups, date pickers, actions, or bulk selection—admin's bundled JS targets specific structure. If we change **ids or classes** the scripts rely on, we may have to rebind behavior ourselves. When in doubt, **diff** our override against the file under `django/contrib/admin/templates/` in our environment rather than starting from a blank page.

2. Using the `admin.site.login_template` attribute

Sometimes, it's even easier to customize just by specifying the template to use in the **admin.py** file. For the login template, there is a specific `admin.site.login_template` attribute that we can use, which allows us to specify a custom template to use for the admin site.

Let's edit the **admin.py** file:

```
admin.site.site_header = "Uber Eat like Admin"
admin.site.index_title = "Welcome back to Uber Eat like
Admin Portal"
admin.site.site_url = None # Remove "View Site" link
admin.site.enable_nav_sidebar = True # Enable the
navigation sidebar
admin.site.login_template = 'ownlogin.html'
```


We don't specify a path for our **ownlogin.html** file, so by default, **Django** will search to the template file relative to the **templates** directory of our project (djangoadmin) and not relative to the **templates/admin** directory of our application (Backoffice) as we learned previously (please refer to Figure 10-4).



Figure 10-4. Admin login template configured via `admin.site.login_template`

And now let's edit our **ownlogin.html** file:

```
...
<form action="{{ app_path }}" method="post" id="login-form">
{% csrf_token %}
  <h4>OWN LOGIN FORM</h4> <!-- OUR MODIFICATION -->
  <div class="form-row">
    {{ form.username.errors }}
    {{ form.username.label_tag }} {{ form.username }}
  </div>
  <div class="form-row">
    {{ form.password.errors }}
    {{ form.password.label_tag }} {{ form.password }}
    <input type="hidden" name="next" value="{{ next }}">
  </div>
{% url 'admin_password_reset' as password_reset_url %}
{% if password_reset_url %}
  <div class="password-reset-link">
```

```

    <a href="{{ password_reset_url }}">{% translate 'Forgotten
    your password or username?' %}</a>
</div>
{% endif %}
<div class="submit-row">
    <input type="submit" value="{% translate 'Log in' %}">
</div>
</form>
...

```

and now we can see our new login form (see Figure 10-5) and not the previous one.

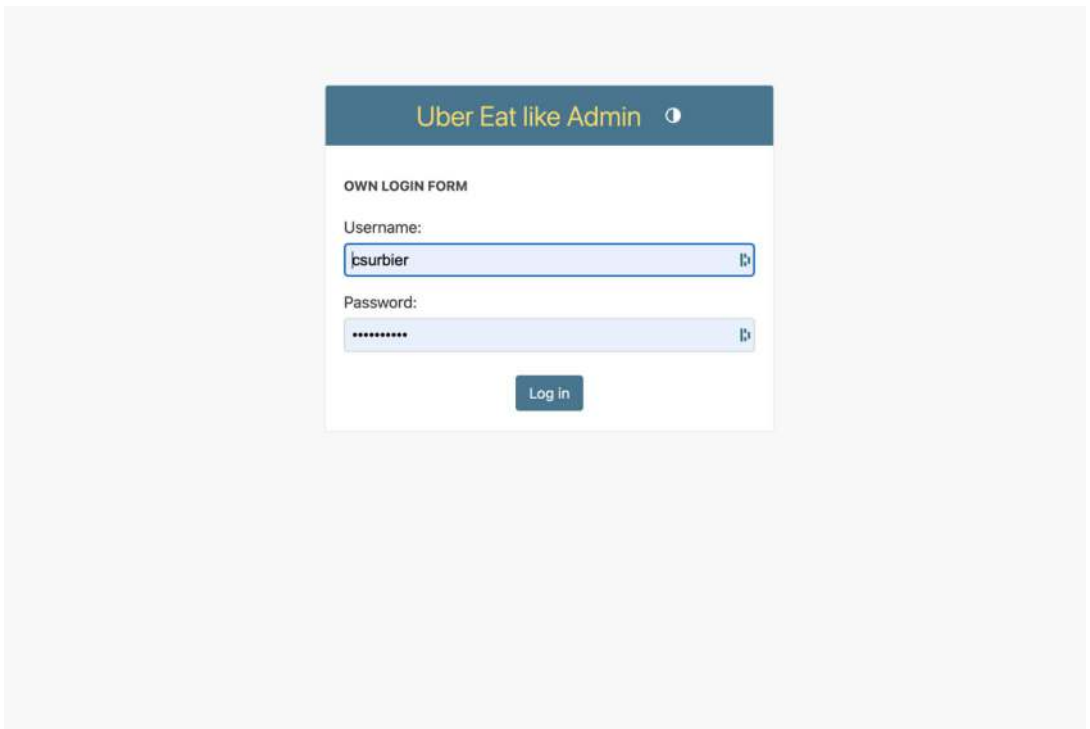


Figure 10-5. Custom standalone login template displayed for the admin site

Change Default Ordering of Our Models in the Admin Site

By default, models displayed in the admin site are ordered by the model name (as shown on Figure 10-6).



Figure 10-6. Default admin model ordering (alphabetical)

To change that ordering, we will override the default `get_app_list` method of the `AdminSite` class.

As a reminder, with Django, we can override lots of default methods.

[Documentation \(https://docs.djangoproject.com/en/5.1/ref/contrib/admin/#adminsite-methods\)](https://docs.djangoproject.com/en/5.1/ref/contrib/admin/#adminsite-methods).

This method **get_app_list** is called when the admin site is loaded to get the list of apps to display. So let's modify our **admin.py** file:

```
admin.site.login_template = 'ownlogin.html'
admin.site.get_app_list = custom_get_app_list
```

The default Django Admin method is the following:

```
def get_app_list(request, app_label=None):
    """
        Return a sorted list of all the installed apps that have been
        registered in this site.
    """
    app_dict = admin.site._build_app_dict(request, app_label)
    # Sort the apps alphabetically.
    app_list = sorted(app_dict.values(), key=lambda x: x["name"].lower())

    # Sort the models alphabetically within each app.
    for app in app_list:
        app["models"].sort(key=lambda x: x["name"])

    return app_list
```

So now, we need to create our **custom_get_app_list** function and modify that code:

```
def custom_get_app_list(request, app_label=None):
    """
        Return a sorted list of all the installed apps that have been
        registered in this site.
    """
    app_dict = admin.site._build_app_dict(request, app_label)
    # Sort the apps alphabetically.
    app_list = sorted(app_dict.values(), key=lambda x: x["name"].lower())
```

```
# Sort the models alphabetically within each app.
for app in app_list:
    app["models"].sort(key=lambda x: x["name"], reverse=True)
    #descending order

return app_list
```

We just need to add the **reverse=True** parameter to our **sort** method to order our models in descending order (Figure 10-7).



Figure 10-7. Custom model ordering in the admin app list (descending)

For ordering based on whatever you want, just modify the `app_list` variable to sort it as you want.

Change Default Admin Index Page

OK now, let's say we would like to change this default admin index page. We will copy the **index.html** file from the **django.contrib.admin** package to our **backoffice/templates/admin** directory. This will override the default **index.html** file:

```
cp ../venv/lib/python3.12/site-packages/django/contrib/admin/templates/
admin/index.html backoffice/templates/admin/
```

And now, we can modify our **index.html** file to add some custom content (see Figure 10-8). To do so, we need to search the **{% block content %}** tag, which is the main block used to display our models list:

```
...
{% block content %}
<div id="content-main">
    MY OWN INDEX
    {% include "admin/app_list.html" with app_list=app_list show_
changelinks=True %}
</div>
{% endblock %}
...
```

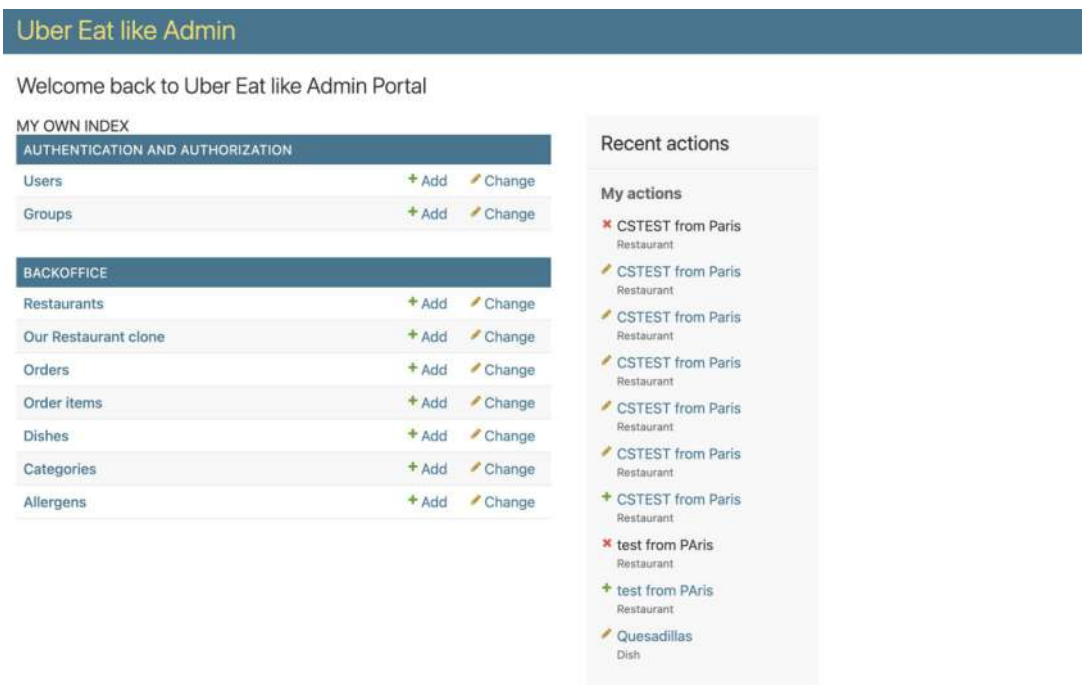


Figure 10-8. Admin index page after overriding `index.html` content block

This `index` method is one of the core view methods in Django’s `AdminSite` class. It’s responsible for rendering the main dashboard page in the Django Admin interface. This page typically displays

- A list of installed apps and their models
- Recent actions taken by the user
- Any additional information provided through the template

The method signature looks like this:

```
def index(self, request, extra_context=None):
    """
    Display the main admin index page, which lists all of the installed
    apps that have been registered in this site.
    """
    # Method implementation...
```

And the `extra_context` parameter is a powerful feature that allows us to inject additional data into the template context without having to completely override the index view. With `extra_context`, we can easily add data that will be passed to the template.

Let's modify our **admin.py** file to specify our own index method:

```
admin.site.index = custom_index
```

And now we can create our **custom_index** method:

```
from datetime import datetime, timedelta

def custom_index(request, extra_context=None):
    # Initialize extra_context if None
    extra_context = extra_context or {}

    # Add custom statistics to the context
    extra_context.update({
        'total_restaurants': Restaurant.objects.count(),
        'active_restaurants': Restaurant.objects.filter(is_active=True).
        count(),
        'total_dishes': Dish.objects.count(),
        'recent_orders': Order.objects.filter(created_at__gte=datetime.
        now() - timedelta(days=7)).count()
    })

    # Call the original index method with our enhanced context
    return admin.site.__class__.index(admin.site, request, extra_context)
```

This method will add some custom statistics to our index page (Figure 10-9).

Note Keep calculations efficient. Since these calculations happen on every dashboard view, make sure they're optimized (consider caching for expensive operations).

And now, we can modify our **index.html** file to display our custom statistics:

```
{% block content %}
<div id="content-main">
  <div class="dashboard-stats" >
    <div class="stat-card">
      <h3>{{ total_restaurants }}</h3>
      <p>Total Restaurants</p>
    </div>
    <div class="stat-card">
      <h3>{{ active_restaurants }}</h3>
      <p>Active Restaurants</p>
    </div>
    <div class="stat-card">
      <h3>{{ total_dishes }}</h3>
      <p>Total dishes</p>
    </div>
    <div class="stat-card">
      <h3>{{ recent_orders }}</h3>
      <p>Total recent orders</p>
    </div>
  </div>
  {% include "admin/app_list.html" with app_list=app_list show_
changelinks=True %}
</div>
{% endblock %}
```

And add some custom CSS for better display. Let's add our CSS stylesheet before the {% endblock %} tag of the block {% block extrastyle %}:

```
{% block extrastyle %}{{ block.super }}<link rel="stylesheet" href="{%
static "admin/css/dashboard.css" %}">
<style>
  .dashboard-stats {
    display: flex;
    flex-wrap: wrap;
```

```

    gap: 20px;
    margin: 20px 0 30px;
}

.stat-card {
    flex: 1;
    min-width: 200px;
    background: white;
    border-radius: 8px;
    box-shadow: 0 4px 12px rgba(0, 0, 0, 0.08);
    padding: 20px;
    text-align: center;
    transition: transform 0.3s ease, box-shadow 0.3s ease;
}

.stat-card:hover {
    transform: translateY(-5px);
    box-shadow: 0 6px 16px rgba(0, 0, 0, 0.12);
}

.stat-card h3 {
    font-size: 32px;
    margin: 0 0 10px;
    color: #417690;
    font-weight: 600;
}

.stat-card p {
    margin: 0;
    color: #666;
    font-size: 14px;
    text-transform: uppercase;
    letter-spacing: 1px;
}

/* Color variations for different cards */
.stat-card:nth-child(1) h3 { color: #4e73df; }
.stat-card:nth-child(2) h3 { color: #1cc88a; }

```

```
.stat-card:nth-child(3) h3 { color: #36b9cc; }
.stat-card:nth-child(4) h3 { color: #f6c23e; }

/* Responsive adjustments */
@media (max-width: 768px) {
  .dashboard-stats {
    flex-direction: column;
  }

  .stat-card {
    margin-bottom: 15px;
  }
}

</style>
{% endblock %}
```

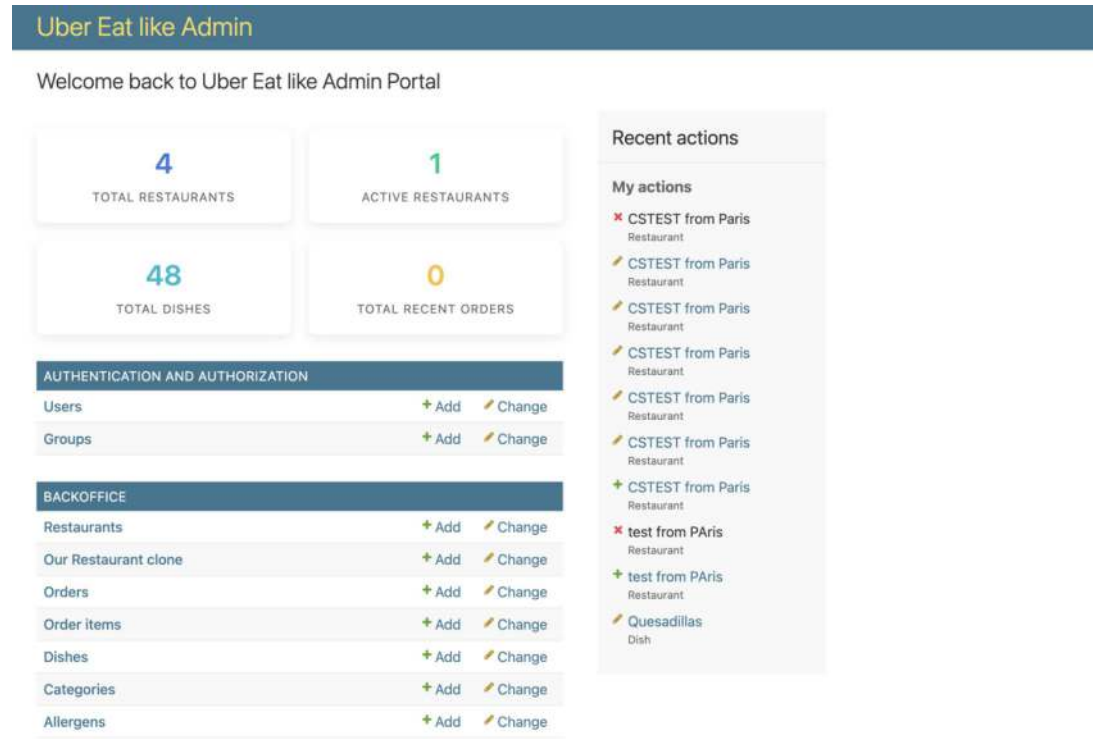


Figure 10-9. Admin index page showing custom dashboard statistics cards (extra_context + custom template)

Change List View

1. Override the `changelist_view` method per model

Not only the index view can be customized, but also the change list view (and other views). The change list view is responsible for displaying a list of objects of a given model, so let's apply the same approach as we did for the index view.

In our **admin.py** file, we can override the **changelist_view** method:

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
    ...
    def changelist_view(self, request, extra_context=None):
        # Initialize extra_context if it's None
        extra_context = extra_context or {}

        extra_context.update({
            'total_orders': Order.objects.
                filter(status="confirmed").count()
        })

        # Custom logic before calling the parent method
        print(f"Accessing changelist for {self.model.__name__}")

        # Call the parent method
        return super().changelist_view(request, extra_context)
```

We are counting the total orders of confirmed status.

Then, we need to override the default **change_list.html** template, but this time, we will do it only for our Restaurant model:

```
mkdir backoffice/templates/admin/backoffice/restaurant/
cp ../venv/lib/python3.12/site-packages/django/contrib/admin/templates/
admin/change_list.html backoffice/templates/admin/backoffice/restaurant
```

Because we want to override for only the **Restaurant** model, we need to create a new folder **restaurant** inside the **backoffice/templates/admin/backoffice** folder and then copy the **change_list.html** file from the **django/contrib/admin/templates/admin** folder to the newly created folder.

And now, we can modify the **change_list.html** file to display our custom statistics (see Figure 10-10):

```
...
<div class="changelist-form-container">
    {% block search %}{% search_form cl %}{% endblock %}
    {% block date_hierarchy %}{% if cl.date_hierarchy %}{% date_
    hierarchy cl %}{% endif %}{% endblock %}
    <!-- OUR ADDITION -->
    <div class="dashboard-stats" >
        <div class="stat-card">
            <p>Total orders:</p>
            <h3>{{total_orders}}</h3>
        </div>
    </div>
```

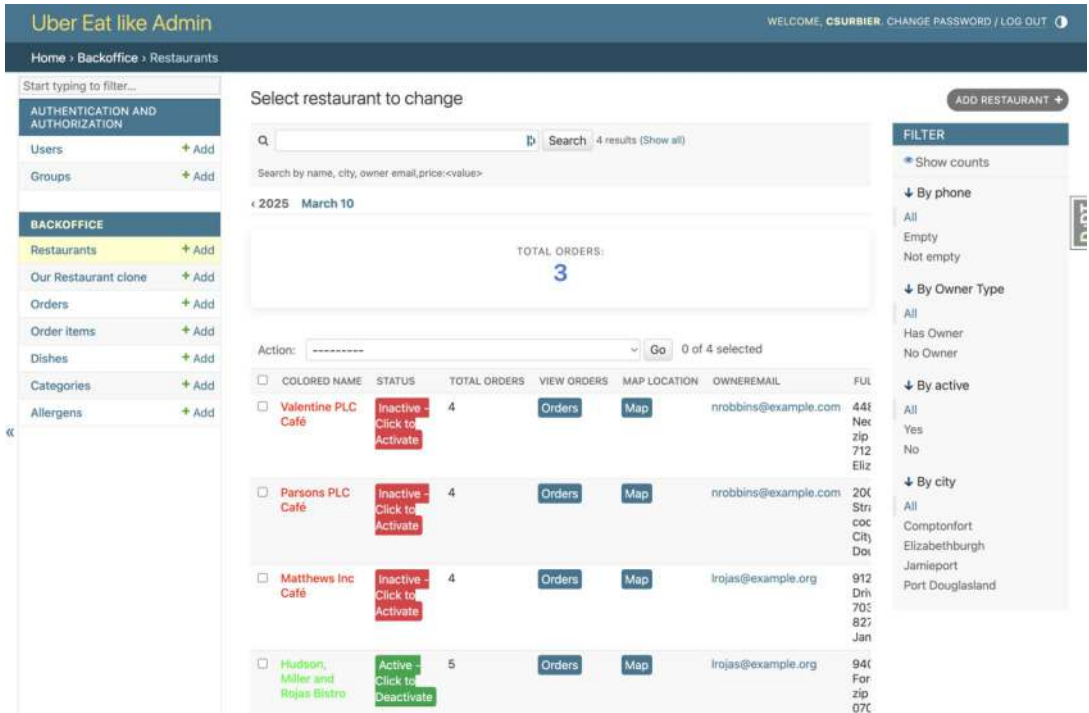


Figure 10-10. Per-model change list customization with extra context rendered in `change_list.html`

We can use the same CSS stylesheet as we did for the index view.

Please notice it is also possible to specify the template path to use for the change list view in the `admin.py` file:

```
class RestaurantAdmin(admin.ModelAdmin):
```

```
...
```

```
change_list_template = 'admin/backoffice/restaurant/change_
list_restaurant.html'
```

Then, we can name it as we want, for example: **`change_list_restaurant.html`**

You can now apply these techniques to customize the admin interface for your own models, on the main methods presented in Table 10-2.

Table 10-2. *ModelAdmin view methods used for add, change, changelist, delete, and history*

Method	Description	Parameters
<code> add_view(request, form_url='', extra_context=None)</code>	Django view for the model instance addition page. Used when creating a new object	request: The HTTP request object form_url: URL to redirect to after successful form submission extra_context: Additional context data for the template
<code> change_view(request, object_id, form_url='', extra_context=None)</code>	Django view for the model instance editing page. Used when modifying an existing object	request: The HTTP request object object_id: The primary key of the object to edit form_url: URL to redirect to after successful form submission extra_context: Additional context data for the template
<code> changelist_view(request, extra_context=None)</code>	Django view for the model instances change list/actions page. Used to display the list of all objects with filtering and actions	request: The HTTP request object extra_context: Additional context data for the template
<code> delete_view(request, object_id, extra_context=None)</code>	Django view for the model instance(s) deletion confirmation page. Used when deleting an object	request: The HTTP request object object_id: The primary key of the object to delete extra_context: Additional context data for the template
<code> history_view(request, object_id, extra_context=None)</code>	Django view for the page that shows the modification history for a given model instance	request: The HTTP request object object_id: The primary key of the object to show history for extra_context: Additional context data for the template

2. Override globally

If we want to apply the same customization to all models, we can override the default html templates and put them in the **backoffice/templates/admin/** folder, but to pass extra context common to all models, we have to create a custom `ModelAdmin` class that inherits from `admin.ModelAdmin` and overrides the desired methods. Then, we can register our custom `ModelAdmin` class with the models you want to customize.

Let's modify our `CategoryAdmin` class to add a global context to its change list:

```
class BaseModelAdmin(admin.ModelAdmin):
    def changelist_view(self, request, extra_context=None):
        # Initialize extra_context if it's None
        extra_context = extra_context or {}

        # Add custom data to the context
        extra_context.update({
            'global_data': 'This data is available in all
                           changelists',
            'model_name': self.model.__name__,
        })

        # Custom logic before calling the parent method
        print(f"Accessing changelist for {self.model.__name__}")

        # Call the parent method
        return super().changelist_view(request, extra_context)

@admin.register(Category)
class CategoryAdmin(BaseModelAdmin):

    list_display = ('name', 'is_active')
    list_filter = ('is_active',)
    search_fields = ('name',)
```

Then, we copy the **change_list.html** template into our **backoffice/templates/admin/** folder:


```
cp ../venv/lib/python3.12/site-packages/django/contrib/admin/
templates/admin/change_list.html backoffice/templates/admin
```

and modify it to display the **global_data** parameter that we added to the context:

```
...
<div class="changelist-form-container">
    {% block search %}{% search_form cl %}{% endblock %}
    {% block date_hierarchy %}{% if cl.date_hierarchy %}{%
    date_hierarchy cl %}{% endif %}{% endblock %}
    <div class="dashboard-stats" >
        <div class="stat-card">
            <p>Global</p>
            <h3>{{global_data}}</h3>

        </div>
    </div>
```

Now, when clicking Category which inherits from the **BaseModelAdmin** class, we can see the global data displayed (Figure 10-11).

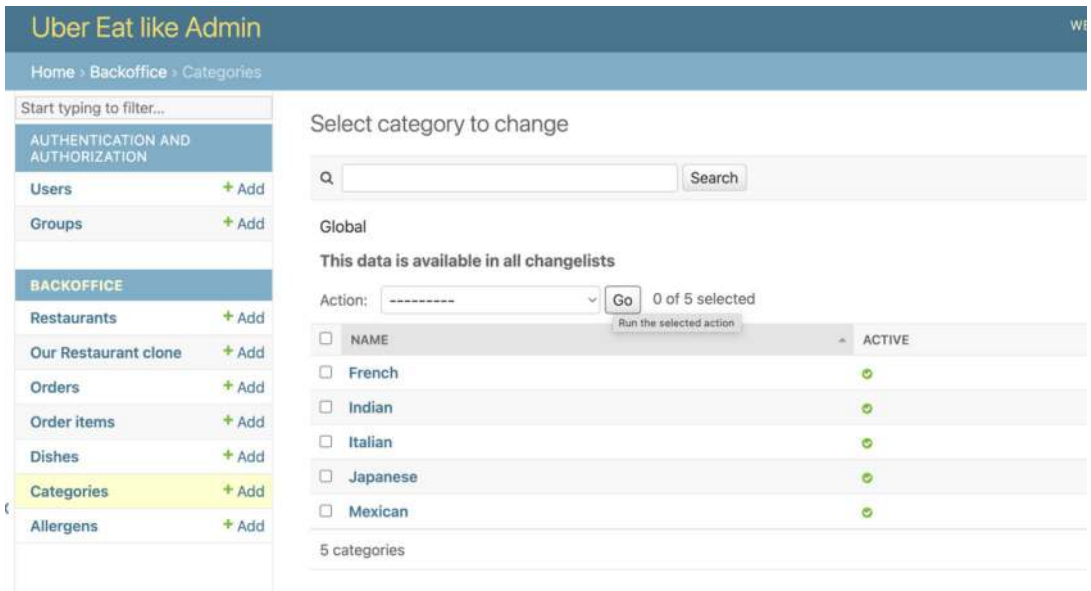


Figure 10-11. Global changelist customization via a shared base *ModelAdmin* and overridden template

Change View: Make an Ajax Request

Another useful feature is to make an Ajax request to the server (or to any other source) to get some data when creating (or editing) a model. If required, you can override the **change_view** of the model to pass extra context as we learned previously.

For now, let's try something simple, when creating a dish, once the user has selected a category, we will display a JavaScript alert with the selected category.

First, we need to create a new folder to store the **change_form.html** template for the dish model that we want to override:

```
mkdir backoffice/templates/admin/backoffice/dish
```

then we copy the default template:

```
cp ../venv/lib/python3.12/site-packages/django/contrib/admin/templates/
admin/change_form.html backoffice/templates/admin/backoffice/dish
```

And now let's edit it to add our custom JavaScript to detect when a user selects a category:

```
{% extends "admin/base_site.html" %}
{% load i18n admin_urls static admin_modify %}

{% block extrahead %}
{{ block.super }}
<script src="{% url 'admin:jsi18n' %}"></script>

<script>
$(document).ready(function() {
    // Add event listener to the category select field
    $('#id_category').change(function() {
        // Get the selected category name
        var categoryId = $(this).val();
        var categoryName = $(this).find('option:selected').text();

        // Display an alert with the selected category
        alert('You selected category: ' + categoryName);
    });
});
</script>
{{ media }}
{% endblock %}
```

1. We added a custom event listener on **#id_category** to detect when the user selects a category.
2. We get the category name and then display an alert with that name.

To know the field name on which we need to set our event listener, we can use the **inspect** tool in the browser (Figure 10-12) to inspect the element and see the id of the select field.

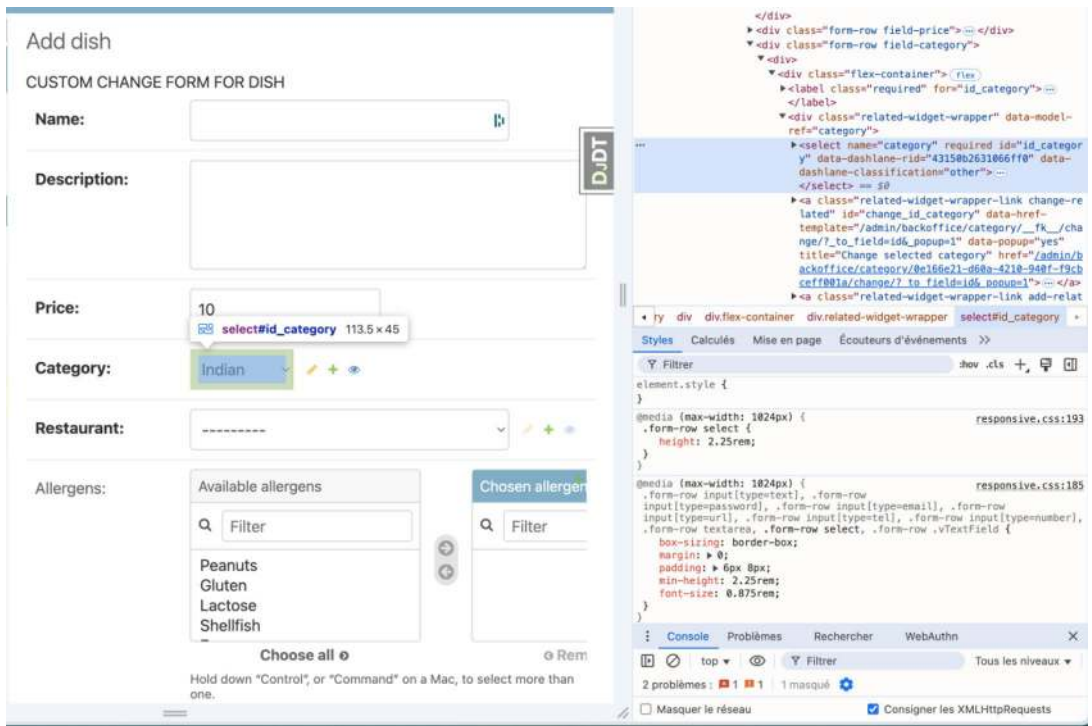


Figure 10-12. Using browser devtools to find the form field selector/id for JavaScript hooks

To test our code is working, we just need to select a category, and we should see the JavaScript alert displayed (Figure 10-13).

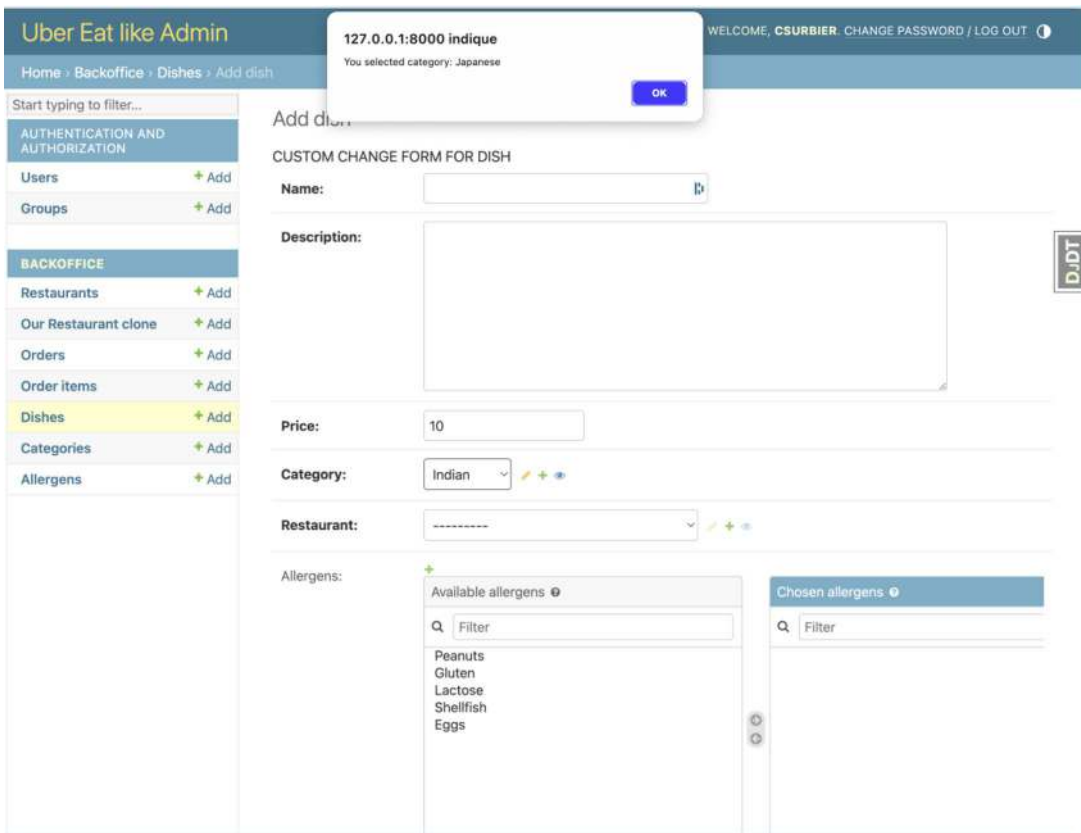


Figure 10-13. Category selection event in the Dish change form (triggering custom JavaScript)

OK now, our goal will be to get the medium price of dishes for the selected category. So we will have to call an API endpoint with the id of the selected category, calculate the medium price, and return the response as a json. Let's implement our API endpoint code.

Edit the **backoffice/views.py** file and add the following code:

```
from django.shortcuts import render
from django.http import JsonResponse
from django.contrib.admin.views.decorators import staff_member_required
from .models import Category, Dish

@staff_member_required
def get_category_dishes(request):
    """
```

Ajax view to get dishes medium prices for a specific category
 """

```
category_id = request.GET.get('category_id')
print("===Category received ", category_id)
if not category_id:
    return JsonResponse({'error': 'No category ID provided'},
                        status=400)

try:
    from django.db.models import Avg
    # Calculate the average price for the dishes in the category
    result = Dish.objects.filter(category_id=category_id).
    aggregate(avg_price=Avg("price"))
    medium_price = float(result["avg_price"]) if result["avg_price"] is
    not None else 0
    # Format the data for JSON response
    data = {
        'medium_price': medium_price
    }

    print(data)
    return JsonResponse({
        'data': data
    })
except Exception as e:
    print("===Error getting medium price for category", e)
    return JsonResponse({'error': str(e)}, status=500)
```

1. We get the category id from the request.
2. We query the database to get the dishes for the category.
3. We calculate the medium price for the category.
4. We return the medium price as a json response.

Now, we need to edit our project **urls.py** file to add the new endpoint:

```
from django.conf import settings
from django.contrib import admin
from django.urls import include, path
from backoffice import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/categories/', views.get_categories, name='get_categories'),
    path('api/get-category-dishes/', views.get_category_dishes, name=
        'get-category-dishes'),
]
```

Finally, we can modify our **change_form.html** JavaScript method, to call this API using an Ajax request:

<script>

```
$(document).ready(function() {
    // Add event listener to the category select field
    $('#id_category').change(function() {
        // Get the selected category name
        var categoryId = $(this).val();
        var categoryName = $(this).find('option:selected').text();

        // Display an alert with the selected category
        //alert('You selected category: ' + categoryName);
        // Make Ajax request to get dishes in the same category
        $.ajax({
            url: '/api/get-category-dishes/',
            type: 'GET',
            data: { 'category_id': categoryId },
            success: function(data) {
                console.log(data);
            },
            error: function(xhr, status, error) {
                // Handle error
            }
        });
    });
});
```

```

        console.log(error)
    }
    });
});
});
</script>

```

Note Since `category_id` is **required** to fetch dishes for a category, a clear alternative is to pass it in the **URL path** (GET `/api/get-category-dishes/{category_id}/`) instead of the Ajax data option. In Django, we would register something like `path('api/get-category-dishes/<uuid:category_id>/', views.get_category_dishes, name='get-category-dishes')`, and in jQuery, we would set `url: '/api/get-category-dishes/' + categoryId + '/'` (and omit data). A path parameter often reads better for a **required** identifier.

Using browser dev tools, we can see in the network tab that when selecting **Italian** category, a request is made to our endpoint (Figure 10-14).

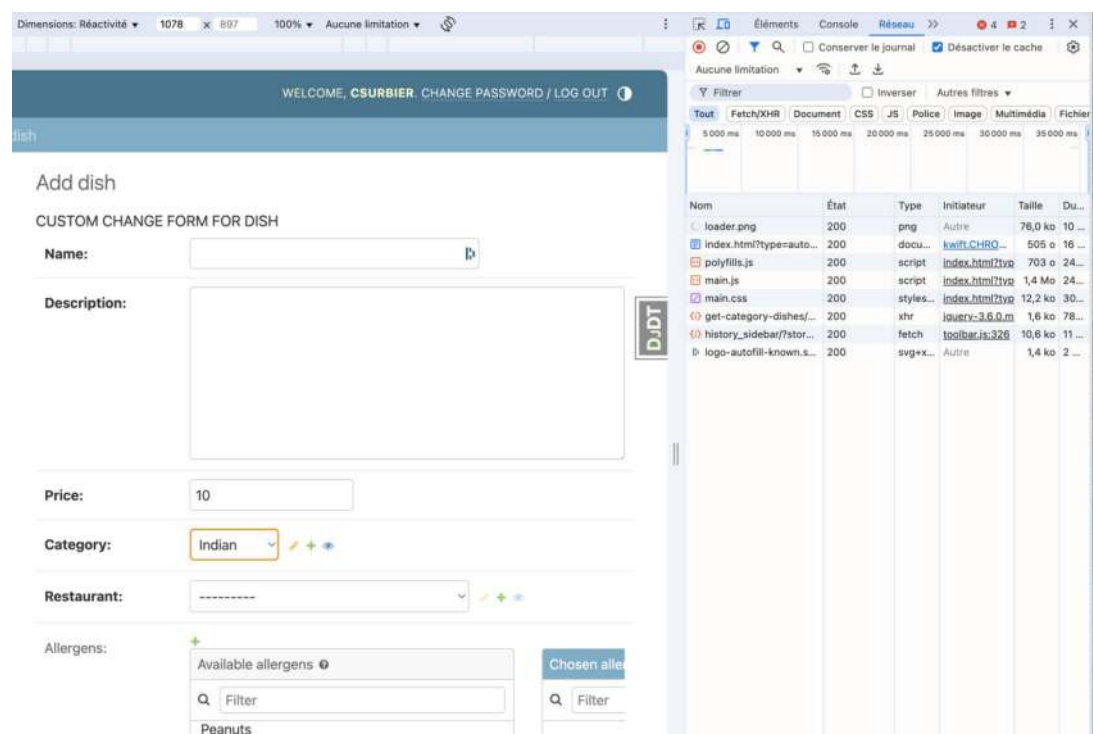


Figure 10-14. Network tab showing the Ajax request triggered by category selection

Inspecting the request response, we can see that the medium price is returned as a json object (Figure 10-15).

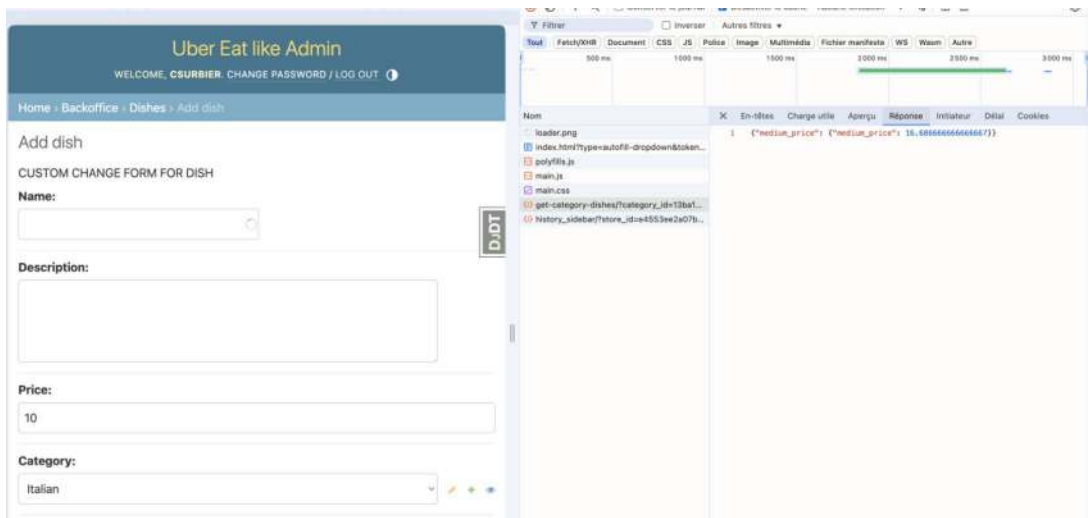


Figure 10-15. Ajax JSON response containing the computed medium price for the category

And data is displayed in the console.log (Figure 10-16).

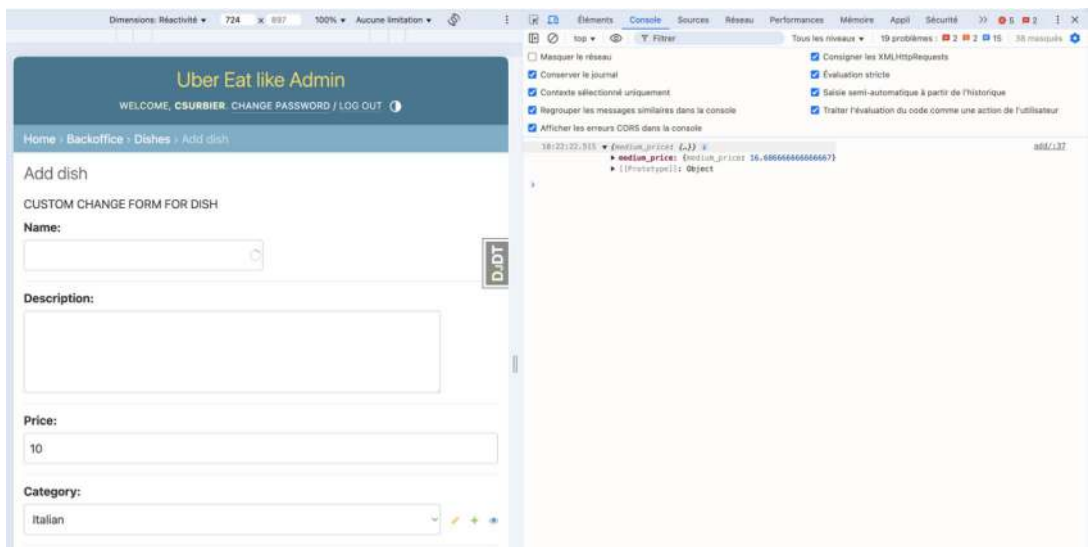


Figure 10-16. Ajax response inspected in the browser console via console.log

So, we will display the medium price in the form to show it to the user, and we will add a custom button to apply the price to the **price** field of the form (as shown on Figure 10-17).

To do that, JavaScript skill is required which I assumed you have.

First, let's style a specific container to display the medium price information:

```
{% extends "admin/base_site.html" %}
{% load i18n admin_urls static admin_modify %}

{% block extrahead %}
{{ block.super }}
<script src="{% url 'admin:jsi18n' %}"></script>
<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
<style>
    .price-recommendation {
        margin-top: 10px;
        padding: 10px;
        border: 1px solid #ddd;
        border-radius: 4px;
        background-color: #f8f9fa;
    }
    .price-recommendation .recommended {
        font-weight: bold;
        color: #28a745;
    }
    .price-recommendation .info-icon {
        color: #17a2b8;
        cursor: pointer;
    }
    .price-recommendation-details {
        margin-top: 10px;
        display: none;
    }
</style>
```

Then, let's update our JavaScript code:

<script>

```
$(document).ready(function() {
    //1. Add a container for price recommendation after the price field
    $('<div id="price-recommendation" class="price-recommendation">
    </div>').insertAfter('.field-price');

    // Add event listener to the category select field
    $('#id_category').change(function() {
        // Get the selected category name
        var categoryId = $(this).val();
        var categoryName = $(this).find('option:selected').text();

        // Display an alert with the selected category
        //alert('You selected category: ' + categoryName);
        //2. Make Ajax request to get dishes in the same category
        $.ajax({
            url: '/api/get-category-dishes/',
            type: 'GET',
            data: { 'category_id': categoryId },
            success: function(data) {
                console.log(data);
                // Format the recommended price to 2 decimal places
                var medium_price = data.data.medium_price.toFixed(2);

                //3. Display the recommended price
                var html = '<div>' +
                    'Based on other dishes for that category' +
                    'the recommended price is <span'
                    class="recommended">$' + medium_price + '</'
                    span></div>' +
                    '<div class="price-recommendation-details">';

                //4. Create a button to apply the price
                html += '</div><button type="button" id="apply-'
                recommended-price" ' +
```

```

        'class="btn btn-sm btn-success mt-2">Apply
        Recommended Price</button>';

    //5. set html
    $('#price-recommendation').html(html);

    //6. Apply recommended price when button is clicked
    $('#apply-recommended-price').click(function() {
        $('#id_price').val(medium_price);
    });
},
error: function(xhr, status, error) {
    // Handle error
    console.log(error)
    // Handle error
    $('#price-recommendation').html('<div class="alert
    alert-danger">' +
        'Error fetching price recommendation: ' + error +
        '</div>');
}
});
});
</script>

```

1. We add a container to display the medium price information.
2. We make our Ajax request to get the medium price of dishes in the selected category.
3. We build a custom HTML to display the recommended price.
4. We create an HTML button to apply the recommended price to the price field.
5. We set our container with the HTML created.
6. When user clicks the button, we set the value to the **price field** of the form (see Figure 10-18).

Uber Eat like Admin

WELCOME, CSURBIER. CHANGE PASSWORD / LOG OUT

Home > Backoffice > Dishes > Add dish

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Users + Add

Groups + Add

BACKOFFICE

Restaurants + Add

Our Restaurant clone + Add

Orders + Add

Order items + Add

Dishes + Add

Categories + Add

Allergens + Add

Add dish

CUSTOM CHANGE FORM FOR DISH

Name:

Description:

Price:

Based on other dishes for that category the recommended price is **\$16.69**

[Apply Recommended Price](#)

Category:

Figure 10-17. Recommended medium price displayed in the admin form UI

Uber Eat like Admin

WELCOME, CSURBIER. CHANGE PASSWORD / LOG OUT

Home > Backoffice > Dishes > Add dish

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Users + Add

Groups + Add

BACKOFFICE

Restaurants + Add

Our Restaurant clone + Add

Orders + Add

Order items + Add

Dishes + Add

Categories + Add

Allergens + Add

Add dish

CUSTOM CHANGE FORM FOR DISH

Name:

Description:

Price:

Based on other dishes for that category the recommended price is **\$16.69**

[Apply Recommended Price](#)

Category:

Restaurant:

Figure 10-18. Recommended price applied to the price field using a custom “Apply” button

Ajax can be used to enhance the Django Admin interface with custom business logic, allowing for more dynamic and interactive admin experiences without requiring page reloads. We use Ajax here to show how admin-side JSON endpoints work; in production, if the values you need are already known when the page is rendered, pass them in template context and expose them to the client with `json_script` (or similar) so you avoid an extra HTTP request; prefer Ajax when the data is too large to ship up front, must be computed after user action, or must stay fresh (live lookups, another user's changes, or heavy aggregates you don't want on every page load).

CHAPTER 11

Custom Views and URLs

Ajax Integration in Admin

To keep our Ajax URL within the admin context, and not write a specific view outside of the admin, to get our data, we can extend the Django Admin URLs.

We just need to register a new URL and an associated view within our **DishAdmin** class as we learned earlier:

```
@admin.register(Dish)
class DishAdmin(admin.ModelAdmin):
    list_display = ('name', 'category', 'restaurant', 'price', 'is_
available',)
    list_filter = (RestaurantFilter, CategoryFilter, 'is_available',)
    list_select_related = ('category', 'restaurant',)
    search_fields = ('name', 'description')
    filter_horizontal = ('allergens',)

    def get_urls(self):
        urls = super().get_urls()
        custom_urls = [
            path('get-mediumprice-for-dishes-category/', self.admin_site.
admin_view(self.get_mediumprice_for_dishes_category),
              name='get-mediumprice-for-dishes-category'),
        ]
        return custom_urls + urls
```

The **get_urls** method is used to add custom URLs to the admin site. The URL added is **get-mediumprice-for-dishes-category**, and it is used to get the dishes' medium price for category.

With that code in place, it means the following URL is available: **/admin/backoffice/dish/get-mediumprice-for-dishes-category/**.

We just need to add our new method `get_mediumprice_for_dishes_category` to the `DishAdmin` class:

```
def get_urls(self):
    urls = super().get_urls()
    custom_urls = [
        path('get-mediumprice-for-dishes-category/', self.admin_site.
            admin_view(self.get_mediumprice_for_dishes_category),
            name='get-mediumprice-for-dishes-category'),
    ]
    return custom_urls + urls

def get_mediumprice_for_dishes_category(self, request):
    """
    Ajax view to get dishes for a specific category and calculate
    average price
    """
    category_id = request.GET.get('category_id')
    if not category_id:
        return JsonResponse({'error': 'No category ID provided'},
            status=400)

    try:
        from django.db.models import Avg
        # Calculate the average price for the dishes in the category
        result = Dish.objects.filter(category_id=category_id).
            aggregate(avg_price=Avg("price"))
        medium_price = float(result["avg_price"]) if result["avg_
            price"] is not None else 0
        # Format the data for JSON response
        data = {
            'medium_price': medium_price
        }
        print("===Getting price for the admin")
        return JsonResponse({
            'data': data
        })
```

except Exception as e:

return JsonResponse({'error': str(e)}, status=500)

and update the **change_form.html** script to call this URL. To do that, we will use the named URL pattern:

```
url: '{% url "admin:get-mediumprice-for-dishes-category" %}'
```

Reminder: **admin:** indicates this URL is in the admin namespace. **get-mediumprice-for-dishes-category** is the specific name of the URL pattern (the one we declared in our `custom_url` method).

And we can use this URL to make our Ajax request and get the data:

```
// Make Ajax request to get dishes in the same category
$.ajax({
    //url: '/api/get-category-dishes/',
    url: '{% url "admin:get-mediumprice-for-dishes-
category" %}',
```

This second approach is more Django-Admin-friendly because

- It uses Django’s admin URL structure.
 - It automatically applies the admin authentication via `admin_site.admin_view`.
 - It keeps all related functionality within the `DishAdmin` class.
-

Faking Models

Before we dive into another implementation of custom view, let me introduce the concept of what “fake models” is. **Fake models** (sometimes called proxy models or virtual models) are representations that appear in the Django Admin interface but don’t directly correspond to database tables (we learned this previously). They serve as interfaces to data that might

- Be calculated or aggregated from multiple models
- Come from external APIs or services
- Represent a specific view of existing data
- Provide a custom interface for complex operations

Unlike regular Django models that map to database tables, fake models don't persist data in the traditional sense. Instead, they act as a presentation layer that allows us to interact with data through the Django Admin interface (see Figure 11-1 and Figure 11-2).

Fake models are particularly useful when

- We need to present data in a way that doesn't match our database structure
- We want to create custom dashboards or reporting interfaces
- We need to provide a simplified interface for complex operations
- We want to integrate external data sources into our admin interface

As example, let's implement a fake model that will provide a grid view of dishes organized by categories for each restaurant. This will give restaurant owners a visual way to manage their menu structure.

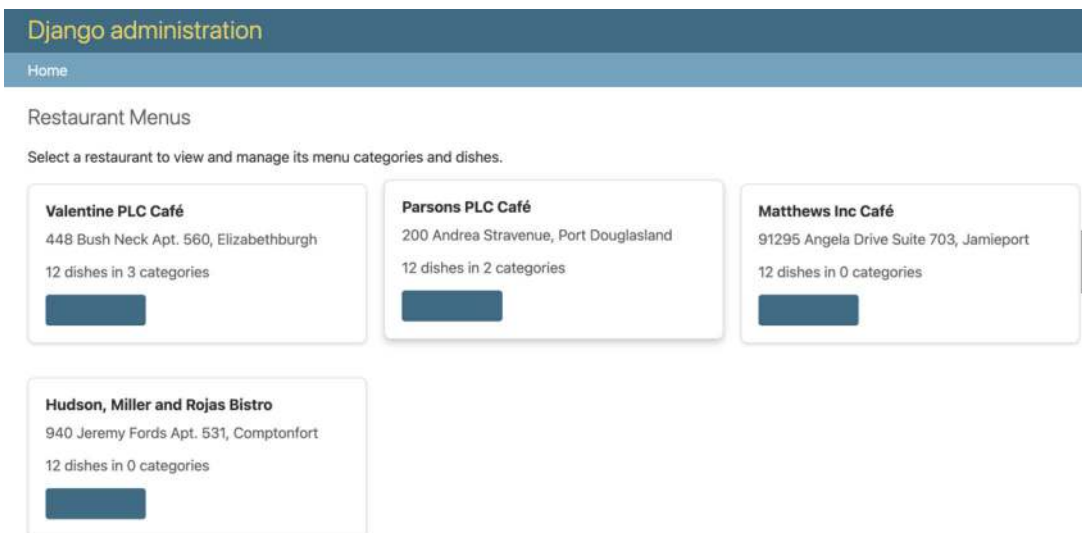


Figure 11-1. Fake/proxy model custom admin page showing a grid of restaurants to manage menus

And when clicking on a specific restaurant, we can display the dishes for each category of the restaurant.

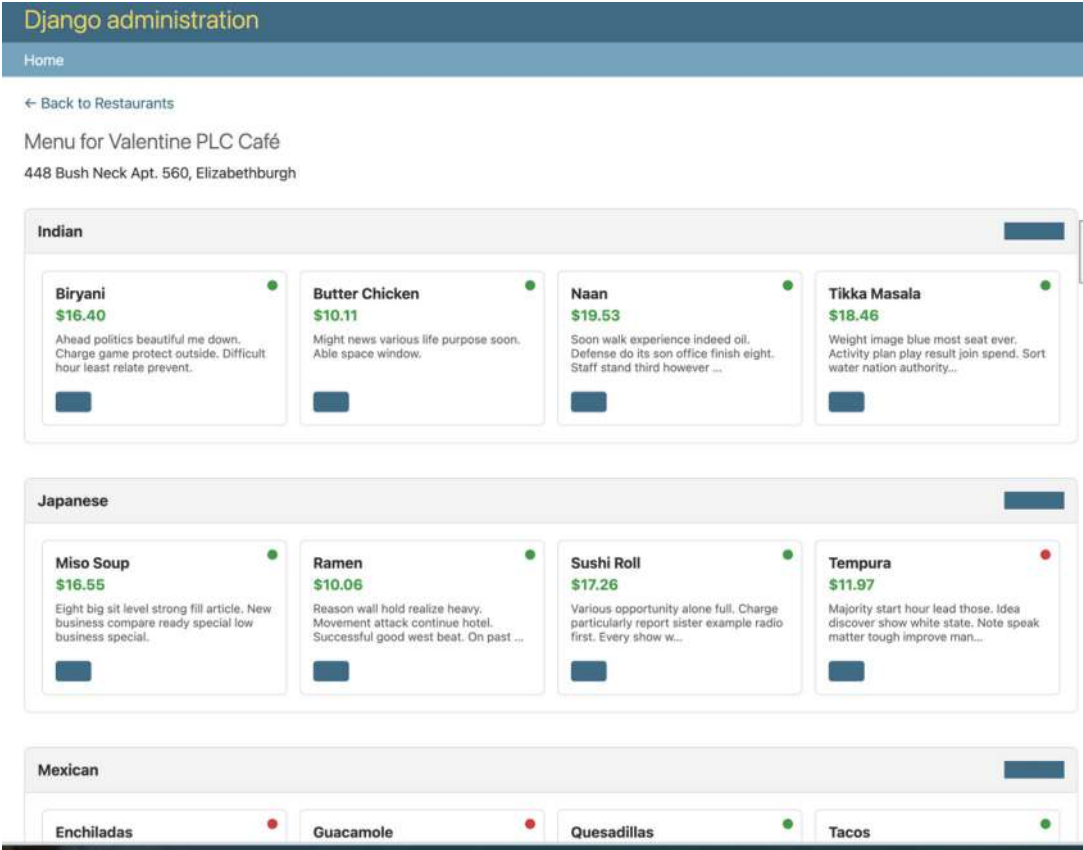


Figure 11-2. Restaurant menu view listing dishes grouped by category

And we can also click an edit button to go directly to the **Dish** change form (Figure 11-3).

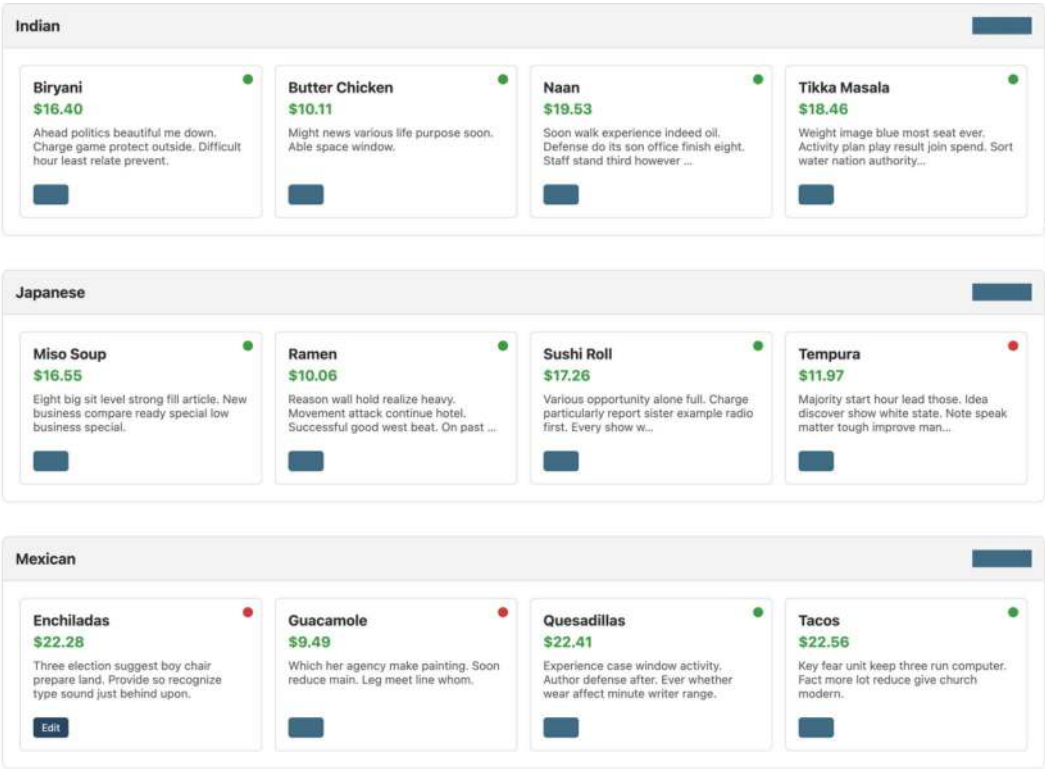


Figure 11-3. Restaurant menu view with direct links to edit dishes in the admin

Of course, the data displayed on your interface will be different and based on your specific generated fake data.

```
Let's start with our fake model for the Restaurant model:
```

```
class RestaurantMenuView(models.Model):  
    """  
    A fake model that doesn't create a database table.  
    Used to provide a custom view in the admin for restaurant menus.  
    """  
  
    id = models.UUIDField(primary_key=True, default=uuid.uuid4,  
        editable=False)  
    restaurant = models.ForeignKey(Restaurant, on_delete=models.DO_NOTHING)
```

```

class Meta:
    managed = False # Tells Django not to create a database table
    verbose_name = "Restaurant Menu View"
    verbose_name_plural = "Restaurant Menu Views"

def __str__(self):
    return f"Menu for {self.restaurant.name}"

```

Reminder: We need to set **managed = False** in the Meta class to tell Django not to create a database table for this model.

Now we will create our admin class for the **RestaurantMenuView** model:

```

@admin.register(RestaurantMenuView)
class RestaurantMenuViewAdmin(admin.ModelAdmin):
    list_display = ('restaurant',)

    def get_queryset(self, request):
        """
        Override the default queryset to return restaurant objects.
        Since this is a fake model, we need to provide the data ourselves.
        """
        # Get all restaurants
        return Restaurant.objects.all().select_related("owner")

    def changelist_view(self, request, extra_context=None):
        """
        Override the default changelist view to display our custom grid.
        """
        # Get the queryset (restaurants)
        restaurants = self.get_queryset(request)

        # Prepare data for the template
        context = {
            'restaurants': restaurants,
        }

```

```
# Return our custom template
return render(request, 'admin/backoffice/restaurantmenuview/menu_
grid.html', context)
```

1. We override the default queryset to return restaurant objects.
2. We override the default changelist view to display our custom grid.
3. We prepare data for the template.
4. We return our custom template.

Let's create our custom **menu_grid.html** template to display our restaurant menus in a grid. Our view will extend the **admin/base_site.html** template to be integrated smoothly in our Django interface:

```
{% extends "admin/base_site.html" %}
{% load i18n admin_urls static admin_list %}

{% block extrastyle %}
  {{ block.super }}
<style>
  .restaurant-card {
    border: 1px solid #ddd;
    border-radius: 8px;
    padding: 20px;
    margin-bottom: 20px;
    box-shadow: 0 2px 4px rgba(0,0,0,0.1);
    transition: transform 0.2s;
  }

  .restaurant-card:hover {
    transform: translateY(-5px);
    box-shadow: 0 4px 8px rgba(0,0,0,0.2);
  }

  .restaurant-grid {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));
    gap: 20px;
```

```

    margin-top: 20px;
}

.restaurant-name {
    font-size: 18px;
    font-weight: bold;
    margin-bottom: 10px;
}

.restaurant-info {
    color: #666;
    margin-bottom: 15px;
}

.view-menu-btn {
    display: inline-block;
    padding: 8px 16px;
    background-color: #417690;
    color: white;
    text-decoration: none;
    border-radius: 4px;
    transition: background-color 0.2s;
}

.view-menu-btn:hover {
    background-color: #2b5070;
    color: white;
}
</style>
{% endblock %}

{% block content %}
<div id="content-main">
    <h1>Restaurant Menus</h1>
    <p>Select a restaurant to view and manage its menu categories and
    dishes.</p>

    <div class="restaurant-grid">
        {% for restaurant in restaurants %}

```



```

<div class="restaurant-card">
  <div class="restaurant-name">{{ restaurant.name }}</div>
  <div class="restaurant-info">
    <p>{{ restaurant.address }}, {{ restaurant.city }}</p>
    <p>{{ restaurant.dishes_restaurant.count }} dishes in {{
      restaurant.categories.count }} categories</p>
  </div>
  <a href="#" class="view-menu-btn">
    View Menu
  </a>
</div>
{% empty %}
  <p>No restaurants available.</p>
{% endfor %}
</div>
</div>
{% endblock %}

```

which will display the screen shown in Figure 11-4.

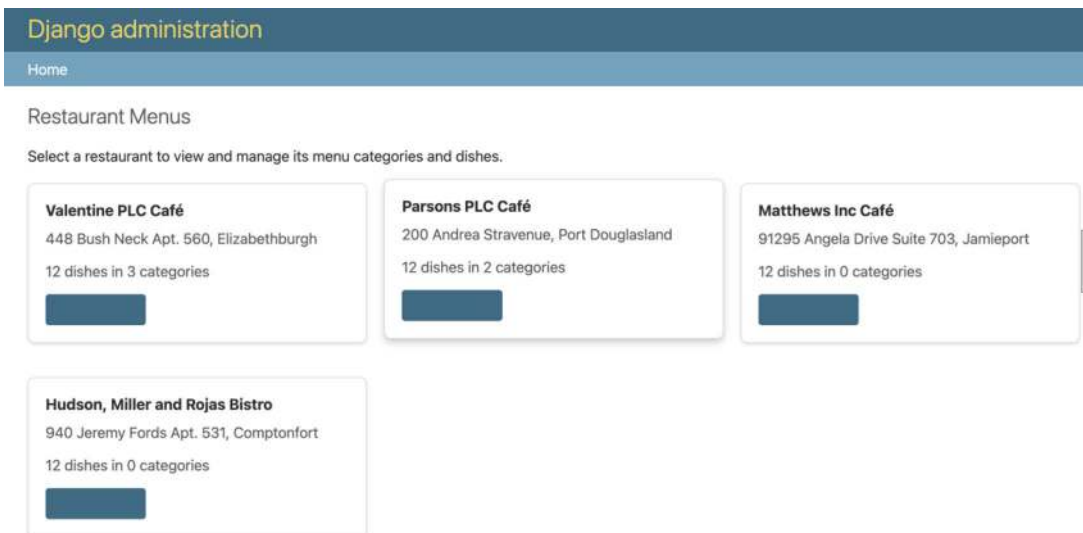


Figure 11-4. Fake/proxy model custom admin page showing a grid of restaurants to manage menus

Next, our goal is when clicking on a Restaurant, we want to display it's menu. To do that, we need to create a custom admin URL associated with a custom view, as we learned previously. We will name this view **view-restaurant-menu** and will need to pass the restaurant identifier to be able to find items and display menu.

Let's complete our admin.py file to add our custom URL and custom view:

```
def get_urls(self):
    urls = super().get_urls()
    custom_urls = [
        path('view_menu/<uuid:restaurant_id>/',
            self.admin_site.admin_view(self.view_menu),
            name='view-restaurant-menu'),
    ]
    return custom_urls + urls

def view_menu(self, request, restaurant_id):
    """
    Custom view to display the menu grid for a specific restaurant
    """
    restaurant = get_object_or_404(Restaurant, id=restaurant_id)

    # Get all categories for this restaurant
    categories = Category.objects.filter(restaurants=restaurant)

    # Get all dishes for this restaurant, organized by category
    categories_with_dishes = []
    for category in categories:
        dishes = Dish.objects.filter(
            restaurant=restaurant,
            category=category
        )

        categories_with_dishes.append({
            'category': category,
            'dishes': dishes
        })
```

```

context = {
    'restaurant': restaurant,
    'categories_with_dishes': categories_with_dishes,
}

return render(
    request,
    'admin/backoffice/restaurantmenuview/menu_detail.html',
    context
)

```

And now we can modify our **menu_grid.html** template to add the custom URL:

```

<a href="{% url 'admin:view-restaurant-menu' restaurant.id %}"
class="view-menu-btn">
    View Menu
</a>

```

Finally, we need to create the new HTML template called **menu_detail.html** and place it in the following location: **templates/admin/backoffice/restaurantmenuview/menu_detail.html**.

```

{% extends "admin/base_site.html" %}
{% load i18n admin_urls static admin_list %}

{% block extrastyle %}
    {{ block.super }}
<style>
    .menu-header {
        margin-bottom: 30px;
    }

    .menu-header h1 {
        margin-bottom: 10px;
    }

    .back-link {
        display: inline-block;
        margin-bottom: 20px;
    }

```

```
.category-section {  
  margin-bottom: 40px;  
  border: 1px solid #ddd;  
  border-radius: 8px;  
  overflow: hidden;  
}  
  
.category-header {  
  background-color: #f5f5f5;  
  padding: 15px;  
  border-bottom: 1px solid #ddd;  
  display: flex;  
  justify-content: space-between;  
  align-items: center;  
}  
  
.category-name {  
  font-size: 18px;  
  font-weight: bold;  
}  
  
.dish-grid {  
  display: grid;  
  grid-template-columns: repeat(auto-fill, minmax(250px, 1fr));  
  gap: 15px;  
  padding: 20px;  
}  
  
.dish-card {  
  border: 1px solid #ddd;  
  border-radius: 6px;  
  padding: 15px;  
  position: relative;  
}  
  
.dish-name {  
  font-weight: bold;
```

```
margin-bottom: 5px;
}

.dish-price {
  color: #28a745;
  font-weight: bold;
  margin-bottom: 10px;
}

.dish-description {
  color: #666;
  font-size: 14px;
  margin-bottom: 10px;
  height: 60px;
  overflow: hidden;
}

.dish-status {
  position: absolute;
  top: 10px;
  right: 10px;
  width: 12px;
  height: 12px;
  border-radius: 50%;
}

.status-available {
  background-color: #28a745;
}

.status-unavailable {
  background-color: #dc3545;
}

.dish-actions {
  display: flex;
  justify-content: space-between;
  margin-top: 10px;
}
```

```

.dish-actions a {
    padding: 5px 10px;
    font-size: 12px;
    text-decoration: none;
    border-radius: 4px;
}

.edit-btn {
    background-color: #417690;
    color: white;
}

.edit-btn:hover {
    background-color: #2b5070;
    color: white;
}
</style>
{% endblock %}

{% block content %}
<div id="content-main">
    <div class="menu-header">
        <a href="{% url 'admin:backoffice_restaurantmenuview_changelist' %}"
        class="back-link">
            &larr; Back to Restaurants
        </a>
        <h1>Menu for {{ restaurant.name }}</h1>
        <p>{{ restaurant.address }}, {{ restaurant.city }}</p>
    </div>

    {% for category_data in categories_with_dishes %}
        <div class="category-section">
            <div class="category-header">
                <div class="category-name">{{ category_data.category.name
            }}</div>

```

```

    <a href="{% url 'admin:backoffice_dish_add' %}?category={{
category_data.category.id }}"&restaurant={{ restaurant.id }}"
class="edit-btn">
    Add Dish
</a>
</div>

<div class="dish-grid">
    {% for dish in category_data.dishes %}
        <div class="dish-card">
            <div class="dish-status {% if dish.is_available %}status-
available{% else %}status-unavailable{% endif %}"></div>
            <div class="dish-name">{{ dish.name }}</div>
            <div class="dish-price">${{ dish.price }}</div>
            <div class="dish-description">{{ dish.
description|truncatechars:100 }}</div>
            <div class="dish-actions">
                <a href="{% url 'admin:backoffice_dish_change' dish.id %}"
class="edit-btn">
                    Edit
                </a>
            </div>
        </div>
    {% empty %}
        <p>No dishes in this category.</p>
    {% endfor %}
</div>
</div>
{% empty %}
    <p>No categories available for this restaurant.</p>
{% endfor %}
</div>
{% endblock %}

```

And voila, see Figure 11-5.

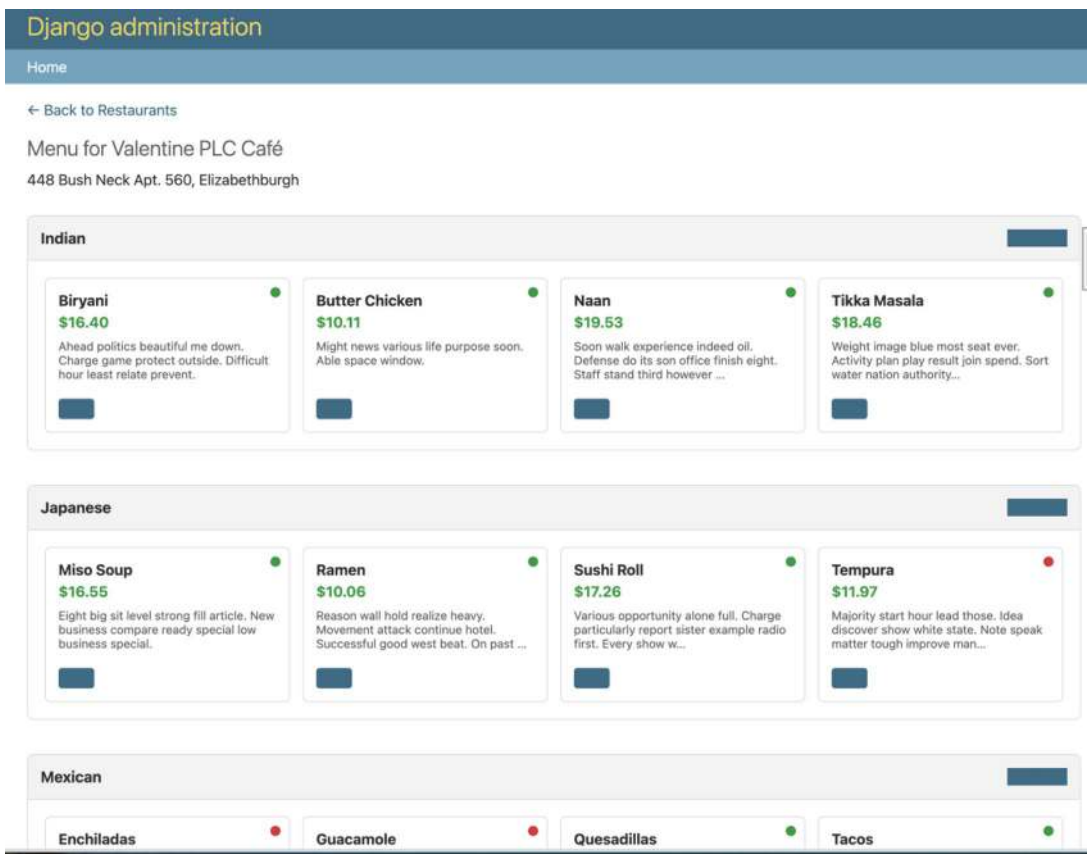


Figure 11-5. Restaurant menu view listing dishes grouped by category

1. We use Django Admin URL reversing to display a link to go back to previous restaurant view `html <a href="{% url 'admin:backoffice_restaurantmenuview_changelist' %}" class="back-link"> ← Back to Restaurants </a>`.
2. We use Django Admin URL reversing to generate the URL for the “Add Dish” link and we passed the selected category and restaurant `html <a href="{% url 'admin:backoffice_dish_add' %}?category={{ category_data.category.id }}&restaurant={{ restaurant.id }}" class="edit-btn"> Add Dish </a>`.

3. We apply the same technique to display a link to edit the Dish
- ```
html <a href="{% url 'admin:backoffice_dish_change'
dish.id %}" class="edit-btn"> Edit .
```

# Advanced URL Management in Django Admin

In the previous example, we had a preview of Django Admin URLs reversing. Let’s dive deeper into URL management. Django Admin’s URL system is powerful and flexible, allowing us to extend the admin interface in sophisticated ways.

Django Admin URLs follow a hierarchical structure as shown in the Table 11-1 below.

*Table 11-1. Default Django Admin URL patterns*

| URL Pattern                                  | Description         |
|----------------------------------------------|---------------------|
| /admin/                                      | Admin index         |
| /admin/<app_label>/                          | App index           |
| /admin/<app_label>/<model_name>/             | Model changelist    |
| /admin/<app_label>/<model_name>/add/         | Add form            |
| /admin/<app_label>/<model_name>/<pk>/change/ | Change form         |
| /admin/<app_label>/<model_name>/<pk>/delete/ | Delete confirmation |

And we’ve also already seen how to add custom URLs to a ModelAdmin class:

```
def get_urls(self):
 urls = super().get_urls()
 custom_urls = [
 path('get-mediumprice-for-dishes-category/',
 self.admin_site.admin_view(self.get_mediumprice_for_dishes_
 category),
 name='get-mediumprice-for-dishes-category'),
]
 return custom_urls + urls
```

## Key Points About Custom URLs

1. **URL Namespacing:** Admin URLs are automatically namespaced with *admin:*, so we can reference them in templates with `{% url "admin:get-mediumprice-for-dishes-category" %}`.
2. **URL Order:** Custom URLs are added before the default URLs (`custom_urls + urls`), ensuring they take precedence over any potentially conflicting patterns.
3. **Security Wrapper:** The `admin_site.admin_view()` wrapper ensures that our view is protected by the same authentication and permission checks as the rest of the admin.

## CHAPTER 12

# REST API Integration with Django Admin

In modern web applications, the admin interface often needs to interact with REST APIs, whether they're internal or external third-party services. Integrating REST APIs with Django Admin enhances its capabilities, allowing us to manage data that might not be directly stored in our database or to interact with external systems.

## API Endpoints

Like we have done with our Ajax request previously, Django Admin can be extended to provide API endpoints that expose administrative functionality programmatically. This is particularly useful for

- Building custom dashboards that need admin data
- Enabling mobile apps to perform admin operations
- Automating administrative tasks
- Supporting rich JavaScript interfaces within the admin

Let's implement a simple API endpoint for our restaurant management system. First, we need to add our custom URLs **api/restaurant-stats** endpoint:

```
def get_urls(self):
 urls = super().get_urls()
 my_urls = [
 path('toggle-active/<path:object_id>/',
 self.admin_site.admin_view(self.toggle_active),
 name='toggle-active-restaurant'),
]
```

```

api_urls = [
 path('api/restaurant-stats/',
 self.admin_site.admin_view(self.restaurant_stats_api),
 name='restaurant-stats-api'),
]
return api_urls+my_urls + urls

```

And then we need to implement the **restaurant\_stats\_api** method in our **RestaurantAdmin** class:

```

from django.db.models import Count
from django.http import JsonResponse
...
def restaurant_stats_api(self, request):
 """API endpoint that returns statistics for restaurants"""
 stats = {
 'total_restaurants': Restaurant.objects.count(),
 'active_restaurants': Restaurant.objects.filter(is_
 active=True).count(),
 'total_orders': Order.objects.count(),
 'recent_orders': Order.objects.filter(
 created_at__gte=datetime.now() - timedelta(days=7)
).count(),
 'top_restaurants': list(
 Restaurant.objects.annotate(
 order_count=Count('orders')
).order_by('-order_count').values('id', 'name',
 'order_count')[:5]
)
 }

 return JsonResponse({'status': 'success', 'data': stats})

```

Our endpoint can be accessed at `/admin/backoffice/restaurant/api/restaurant-stats/` and will return JSON data with restaurant statistics (Figure 12-1).

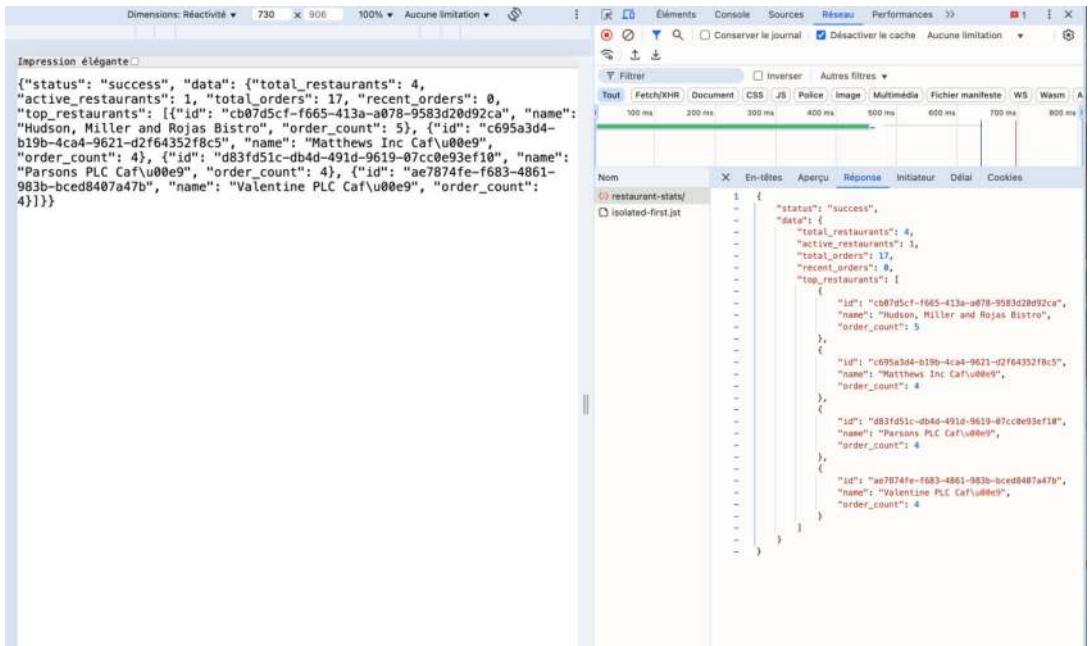


**Figure 12-1.** Custom admin API endpoint returning restaurant statistics as JSON

**Note** Using the same idea, we can call external APIs, process the data, and return whatever we want. Let's imagine we have an external delivery API to know the status of a delivery; we could write something like this:

**try:**

```

response = requests.get(
 f"{settings.DELIVERY_API_URL}/tracking/{order.tracking_number}/",
 headers={"Authorization": f"Bearer {settings.DELIVERY_API_KEY}"})
response.raise_for_status()
tracking_data = response.json()

context = {
 'title': f'Delivery Tracking for Order #{order.id}',
 'order': order,
 'tracking_data': tracking_data,

```

```

 'opts': self.model._meta,
 }

 return render(request, 'admin/backoffice/order/delivery_tracking.html',
context)

except requests.RequestException as e:
 messages.error(request, f"Error fetching tracking data: {str(e)}")
 return redirect('admin:backoffice_order_change', order.id)

```

## Serialization

Django DRF (<https://www.django-rest-framework.org/>) is the standard tool for building APIs in Django. We can integrate with Django Admin to provide powerful serialization capabilities.

Let's add it to our requirements.txt file:

```
django-rest-framework==3.15.2
```

and install it:

```
pip install django-rest-framework==3.15.2
```

Then, we will create a **serializers.py** file in our app:

```
touch backoffice/serializers.py
```

and implement it:

```

from rest_framework import serializers
from .models import Restaurant, Category, Dish

class DishSerializer(serializers.ModelSerializer):
 class Meta:
 model = Dish
 fields = ['id', 'name', 'description', 'price', 'is_available']

class CategorySerializer(serializers.ModelSerializer):
 dishes = DishSerializer(many=True, read_only=True, source='dishes_category')

```

```

class Meta:
 model = Category
 fields = ['id', 'name', 'dishes']

class RestaurantSerializer(serializers.ModelSerializer):
 categories = CategorySerializer(many=True, read_only=True)

class Meta:
 model = Restaurant
 fields = ['id', 'name', 'categories']

```

Our serializers are the JSON representation of our models: **Dish**, **Category**, and **Restaurant**. And using the **fields** attribute, we can control what we want to expose (fields returned in the JSON response):

1. **DishSerializer**: This serializer is used to serialize the **Dish** model. It includes fields like **id**, **name**, **description**, **price**, and **is\_available**.
2. **CategorySerializer**: This serializer is used to serialize the **Category** model. It includes fields like **id**, **name**, and a nested serializer for **dishes**.
3. **RestaurantSerializer**: This serializer is used to serialize the **Restaurant** model. It includes fields like **id**, **name**, and a nested serializer for **categories**.

---

A nested serializer is a serializer that contains other serializers as fields. In this case, the **CategorySerializer** contains a nested serializer for **dishes**, and the **RestaurantSerializer** contains a nested serializer for **categories**. It will make more sense in few minutes when we will use them in our view.

---

So let's use these serializers in our admin views. We will modify our custom URLs once again to add a new endpoint:

```
def get_urls(self):
 urls = super().get_urls()
 my_urls = [
 path('toggle-active/<path:object_id>/',
 self.admin_site.admin_view(self.toggle_active),
 name='toggle-active-restaurant'),
]
 api_urls = [
 path('api/restaurant-stats/',
 self.admin_site.admin_view(self.restaurant_stats_api),
 name='restaurant-stats-api'),
 path('api/menu/<uuid:restaurant_id>/',
 self.admin_site.admin_view(self.restaurant_menu_api),
 name='restaurant-menu-api'),
]
 return api_urls+my_urls + urls
```

and then write the view which will use the **RestaurantSerializer** to return data:

```
def restaurant_menu_api(self, request, restaurant_id):
 """API endpoint that returns the complete menu for a restaurant"""
 restaurant = get_object_or_404(Restaurant, id=restaurant_id)
 serializer = RestaurantSerializer(restaurant)
 return JsonResponse(serializer.data)
```

Now, if we call this new endpoint/API with an existing restaurant id, we will get our JSON data (see Figure 12-2):

```
http://127.0.0.1:8000/admin/backoffice/restaurant/api/menu/ae7874fe-
f683-4861-983b-bced8407a47b/.
```



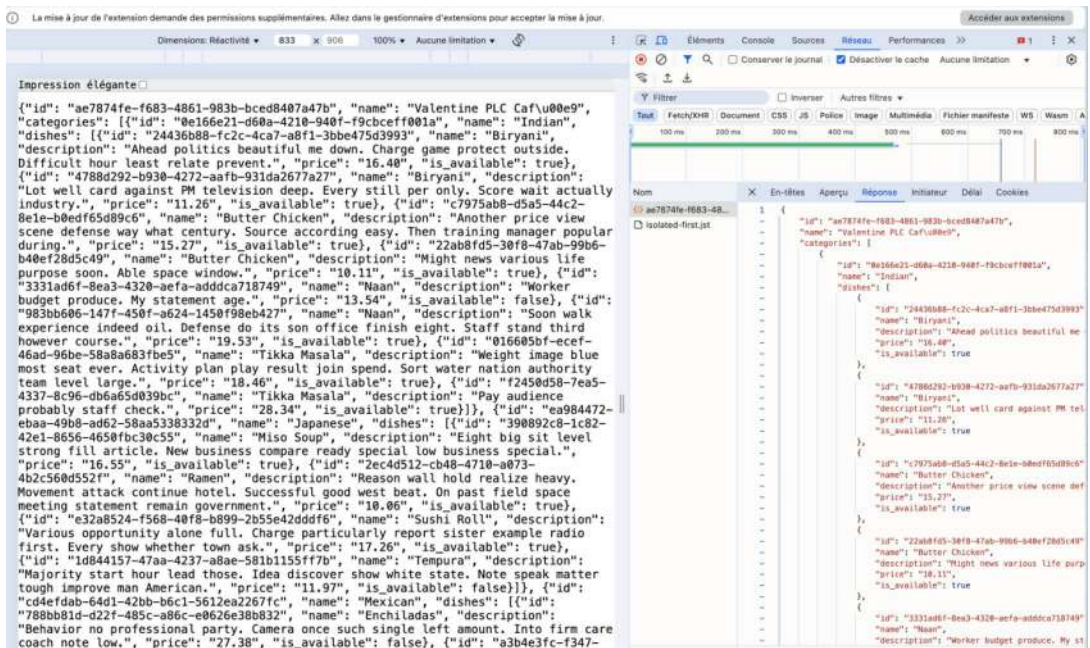


**Figure 12-2.** Custom admin API endpoint returning a restaurant menu via nested serializers

As we can see in the results, we have the categories details as well as the dishes details. Without the nested serializers, we would have had just identifiers of the model. If we removed the nested `CategorySerializer` like this:

```
class RestaurantSerializer(serializers.ModelSerializer):
 #categories = CategorySerializer(many=True, read_only=True)

 class Meta:
 model = Restaurant
 fields = ['id', 'name', 'categories']
```

we would get

```
{
 "id": "ae7874fe-f683-4861-983b-bced8407a47b",
 "name": "Valentine PLC Caf\u00e9",
 "categories": [
 "0e166e21-d60a-4210-940f-f9cbceff001a",
```

```
 "ea984472-ebaa-49b8-ad62-58aa5338332d",
 "cd4efdab-64d1-42bb-b6c1-5612ea2267fc"
]
}
```

**Django DRF** framework is a powerful tool for building APIs. Don't hesitate to refer to [documentation](https://www.django-rest-framework.org/) (<https://www.django-rest-framework.org/>) to learn more.

## CHAPTER 13

# Background Tasks

After API integration, another important aspect is handling background tasks. These are tasks that need to be performed in the background, such as sending emails, processing large datasets, or running scheduled jobs. Django provides a powerful system for handling these tasks using a package called Celery.

## Celery Integration

**Celery** (<https://docs.celeryq.dev/en/stable/>) is the most popular task queue for Django applications, allowing us to run time-consuming tasks asynchronously. Integrating Celery with Django Admin enables to trigger complex operations without waiting for them to complete.

**Redis** (<https://redis.io/>) is a fast and scalable in-memory data store that is commonly used as a message broker for Celery.

Let's add them to our **requirements.txt** file:

```
celery==5.4.0
redis==5.0.7
```

and install them:

```
pip install celery==5.4.0 redis==5.0.7
```

We need to create a **celery.py** file in our project directory:

```
touch djangoadmin/celery.py
```

and edit it:

```
import os
from celery import Celery

Set the default Django settings module
```

```

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'djangoadmin.settings')

Create the Celery app
app = Celery('djangoadmin')

Load configuration from Django settings
app.config_from_object('django.conf:settings', namespace='CELERY')

Auto-discover tasks in all installed apps
app.autodiscover_tasks()
app.conf.timezone = 'Europe/Paris'
app.conf.enable_utc = True

@app.task(bind=True)
def debug_task(self):
 print(f'Request: {self.request!r}')

```

And we need to ensure Celery is loaded when Django starts. For that, we need to edit the `**__init__.py**` file inside our **djangoadmin** project folder:

```

from .celery import app as celery_app

__all__ = ('celery_app',)

```

We will need a **REDIS** server running to use Celery. We can download and configure it locally on our machine or use a free cloud version (see Figure 13-1) using [Redis cloud](https://cloud.redis.io/) (<https://cloud.redis.io/>).

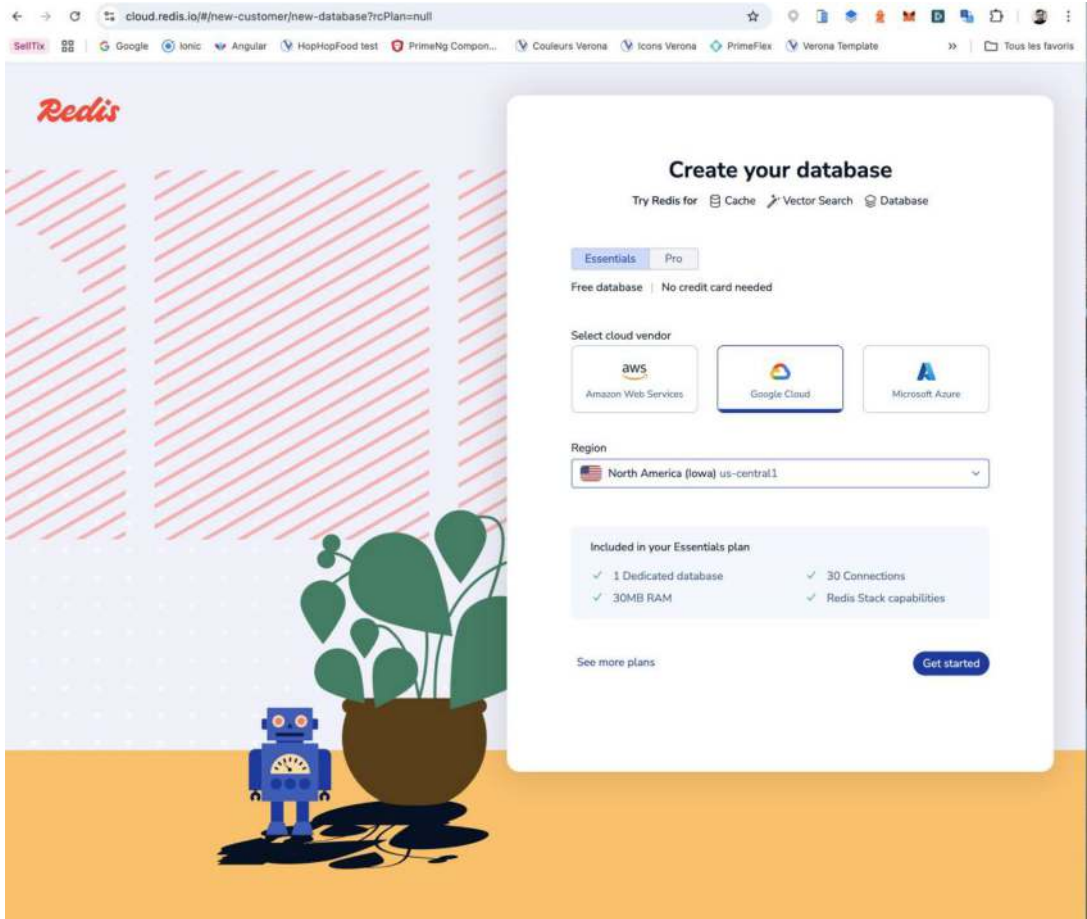


**Figure 13-1.** Redis Cloud setup used as the Celery broker/result backend

---

From now, I will assume you have a configured Redis server running.

---

Let's add our REDIS configuration to our **settings.py** file:

#### *# Celery Configuration*

```
CELERY_BROKER_URL = os.getenv("REDIS_BROKER_URL")+'/'
CELERY_RESULT_BACKEND = os.getenv("REDIS_BROKER_URL")+'/'
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TIMEZONE = "Europe/Paris"
```

If you want to monitor the Redis data, you can use the [RedisInsight](https://redis.io/insight/) (<https://redis.io/insight/>) tool. You can connect with your account, and if you choose a Redis database using Redis Cloud, it will import it automatically.

Let's create a **tasks.py** file in which we will create a long running task which will create a CSV file with restaurant information:

```
from celery import shared_task
from django.core.mail import send_mail
from .models import Restaurant, Order, Dish
import csv
import io
import time
from datetime import datetime, timedelta

@shared_task
def generate_restaurant_report(restaurant_id):
 """Generate a comprehensive report for a restaurant"""
 try:
 # Simulate a time-consuming task
 time.sleep(5)

 restaurant = Restaurant.objects.get(id=restaurant_id)

 # Create a CSV report in a file
 output = io.StringIO()
 writer = csv.writer(output)

 # Write header
 writer.writerow(['Report for', restaurant.name])
 writer.writerow(['Generated at', datetime.now().strftime('%Y-%m-%d %H:%M:%S')])
 writer.writerow([])

 # Write dish statistics
 writer.writerow(['Dish Statistics'])
 writer.writerow(['Name', 'Price', 'Available', 'Category'])

 dishes = Dish.objects.filter(restaurant=restaurant)
 .select_related('category')
```

```

 .select_related('restaurant')
 .prefetch_related("allergens")

for dish in dishes:
 writer.writerow([
 dish.name,
 dish.price,
 'Yes' if dish.is_available else 'No',
 dish.category.name
])

writer.writerow([])

Write order statistics
writer.writerow(['Order Statistics (Last 30 days)'])
writer.writerow(['Date', 'Customer', 'Total Amount', 'Status'])

thirty_days_ago = datetime.now() - timedelta(days=30)
orders = Order.objects.filter(
 restaurant=restaurant,
 created_at__gte=thirty_days_ago
).select_related('customer').select_related('restaurant')

for order in orders:
 writer.writerow([
 order.created_at.strftime('%Y-%m-%d'),
 order.customer.get_full_name(),
 order.total_amount,
 order.status
])

create csv file
with open(f'restaurant_report_{restaurant.id}.csv', 'w') as
csv_file:
 csv_file.write(output.getvalue())
print("===Report generated successfully===")
print(f'restaurant_report_{restaurant.id}.csv')

```

```

except Exception as e:
 return {
 'status': 'error',
 'message': str(e)
 }

```

Now, as we learned, we will add a custom action to our admin, which will call this task:

```

from .tasks import generate_restaurant_report

@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
 form = RestaurantForm
 actions = ['make_inactive', 'export_to_csv', 'generate_report']

 @admin.action(description="Generate a long running task for our
 Restaurant")
 def generate_report(self, request, queryset):
 for restaurant in queryset:
 # Launch the task asynchronously
 task = generate_restaurant_report.delay(
 str(restaurant.id)
)

 self.message_user(
 request,
 f"Report generation started for {queryset.count()}
 restaurants. ",
 messages.SUCCESS
)
 return HttpResponseRedirect(request.get_full_path())

```

---

The important part is to launch our task asynchronously using the **.delay()** method.

---



Now, we need to launch the Celery worker. A celery worker is a process that will check and execute the tasks we define in our code. It will use the Redis database to store the tasks and their status:

```
celery -A djangoadmin worker --loglevel=info
```

You should see the connection information and the available tasks registered with Celery:

```
celery -A djangoadmin worker --loglevel=info

----- celery@csurbier.local v5.4.0 (opalescent)
--- ***** -----
-- ***** ---- macOS-15.3.2-arm64-arm-64bit 2025-04-08 09:28:10
- *** --- * ---
- ** ----- [config]
- ** ----- .> app: djangoadmin:0x1045b27d0
- ** ----- .> transport: redis://default:**@redis-11592.c335.europe-
west2-1.gce.redns.redis-cloud.com:11592/0
- ** ----- .> results: redis://default:**@redis-11592.c335.europe-
west2-1.gce.redns.redis-cloud.com:11592/0
- *** --- * --- .> concurrency: 8 (prefork)
-- ***** ---- .> task events: OFF (enable -E to monitor tasks in
this worker)
--- ***** -----
----- [queues]
 .> celery exchange=celery(direct) key=celery

[tasks]
 . djangoadmin.celery.debug_task
 . backoffice.tasks.generate_restaurant_report
```

Now we can select some restaurants and click on our new custom action **Generate a long running task for our Restaurant** (Figure 13-2).

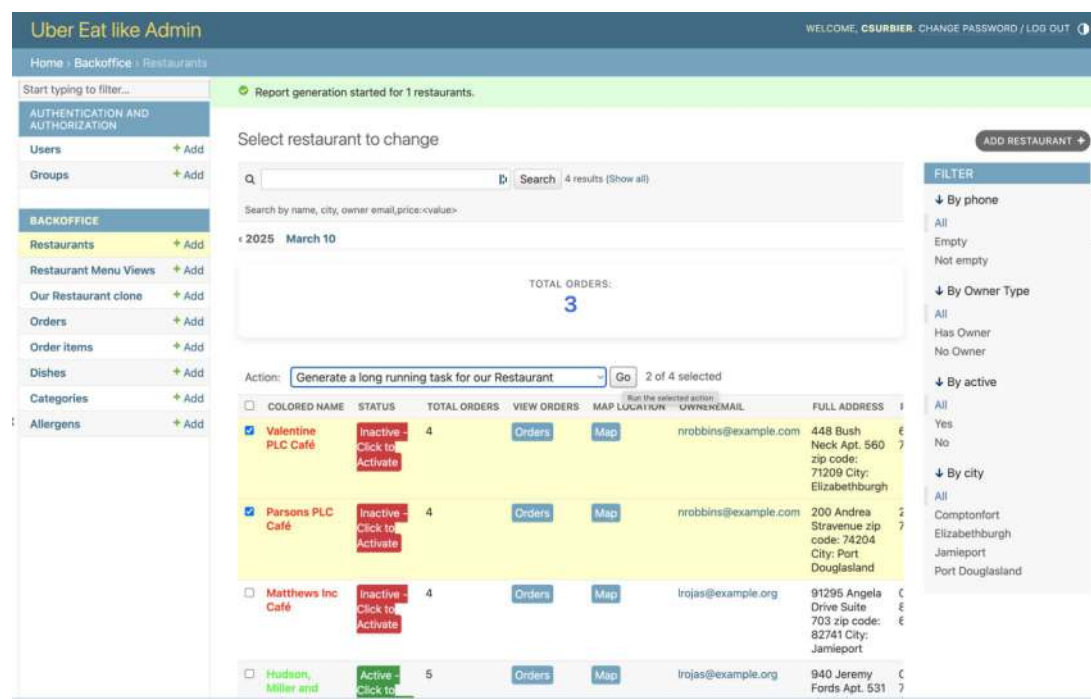


**Figure 13-2.** Triggering a long-running Celery task from a Django Admin action

Our tasks should appear in the logs of the worker:

```
[2025-04-08 09:35:04,417: WARNING/ForkPoolWorker-8] ===Report generated successfully===
[2025-04-08 09:35:04,417: WARNING/ForkPoolWorker-8] restaurant_report_ae7874fe-f683-4861-983b-bced8407a47b.csv
[2025-04-08 09:35:04,456: WARNING/ForkPoolWorker-1] /Users/christophesurbier/.pyenv/versions/3.10.0/lib/python3.10/site-packages/django/db/models/fields/__init__.py:1595: RuntimeWarning: DateTimeField Order.created_at received a naive datetime (2025-03-09 09:35:04.454588) while time zone support is active.
 warnings.warn(
[2025-04-08 09:35:04,484: WARNING/ForkPoolWorker-1] ===Report generated successfully===
[2025-04-08 09:35:04,485: WARNING/ForkPoolWorker-1] restaurant_report_d83fd51c-db4d-491d-9619-07cc0e93ef10.csv
```

```
[2025-04-08 09:35:04,637: INFO/ForkPoolWorker-8] Task backoffice.tasks.
generate_restaurant_report[9b6693ee-c454-4ab9-8cb0-02c674da13bc] succeeded
in 5.582689207978547s: None
```

```
[2025-04-08 09:35:04,715: INFO/ForkPoolWorker-1] Task backoffice.tasks.
generate_restaurant_report[acd3fa1e-eb38-45c0-92e7-866305ef50f4] succeeded
in 5.583825125009753s: None
```

or could be visualized in the **Redis Insight** software (see Figure 13-3).

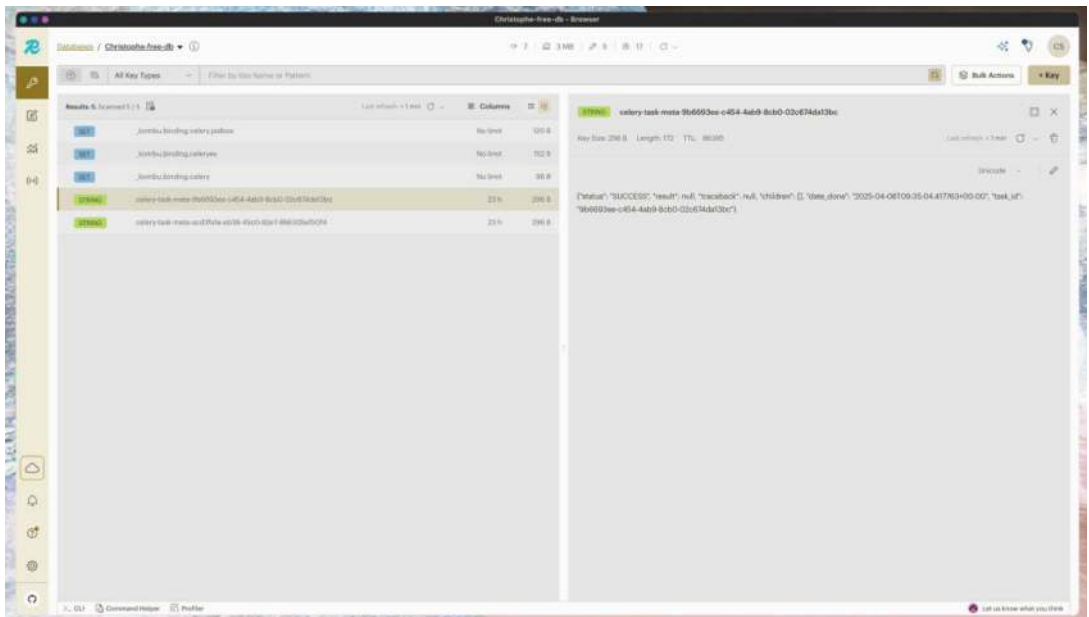


**Figure 13-3.** Monitoring queued/completed tasks using RedisInsight

And of course, we should find our CSV files inside our project directory.

---

You can apply the same technique to any long-running task you want to run asynchronously in your Django application. Just create your task and use the `.delay()` method to call it.

---

## Task Management

Now that we know how to combine Celery and Redis to run asynchronous tasks, we could build a task management system to track the status of our tasks.

We could add a new **BackgroundTask** model to track the status of our tasks:

```
class BackgroundTask(models.Model):
 TASK_STATUS_CHOICES = [
 ('pending', 'Pending'),
 ('running', 'Running'),
 ('completed', 'Completed'),
 ('failed', 'Failed'),
]

 task_id = models.CharField(max_length=255, unique=True)
 task_name = models.CharField(max_length=255)
 status = models.CharField(max_length=20, choices=TASK_STATUS_CHOICES,
 default='pending')
 created_by = models.ForeignKey(User, on_delete=models.SET_NULL,
 null=True)
 created_at = models.DateTimeField(auto_now_add=True)
 completed_at = models.DateTimeField(null=True, blank=True)
 result = models.JSONField(null=True, blank=True)
 error_message = models.TextField(blank=True)

 class Meta:
 ordering = ['-created_at']
 verbose_name = 'Background Task'
 verbose_name_plural = 'Background Tasks'

 def __str__(self):
 return f"{self.task_name} ({self.task_id})"
```

And then use this model with our admin interface to show tasks lifecycle evolution.

Let's first run our migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Then, let's add an admin interface for our BackgroundTask:

```
@admin.register(BackgroundTask)
class BackgroundTaskAdmin(admin.ModelAdmin):
 list_display = ('task_name', 'status', 'created_by', 'created_at',
 'completed_at', 'refresh')
 list_filter = ('status', 'task_name', 'created_at')
 search_fields = ('task_id', 'task_name', 'created_by__username')

 # add a refresh Button
 def refresh(self, obj):
 #refresh admin page
 return format_html('Refresh',
 reverse('admin:backoffice_backgroundtask_changelist'))
 refresh.short_description = 'Refresh'
 refresh.allow_tags = True
```

We added a custom **refresh** button to reload the page on demand.

Now we can modify our **generate\_restaurant\_report** task to create the task at the beginning of the process, wait 20 seconds, do the job, and then update the task status:

```
@shared_task
def generate_restaurant_report(restaurant_id):
 """Generate a comprehensive report for a restaurant"""
 try:
 # Simulate a time-consuming task
 # Create a task record
 task_record = BackgroundTask.objects.create(
 task_id=f"task_{restaurant_id}",
 task_name=f"Restaurant Report - {restaurant_id}",
 status='running'
)
 time.sleep(20)

 restaurant = Restaurant.objects.get(id=restaurant_id)
 ...
 print("===Report generated successfully===")
 print(f'restaurant_report_{restaurant.id}.csv')
```

```
Update task record on success
from django.utils import timezone
task_record.status = 'completed'
task_record.completed_at = timezone.now()
task_record.result = {
 'restaurant_id': restaurant_id,
 'CSV': 'restaurant_report_{restaurant.id}.csv'
}
task_record.save()

except Exception as e:
 return {
 'status': 'error',
 'message': str(e)
 }
```

Now, after launching a new report generation, we could monitor the status of the task (as shown in Figure 13-4), using our admin.

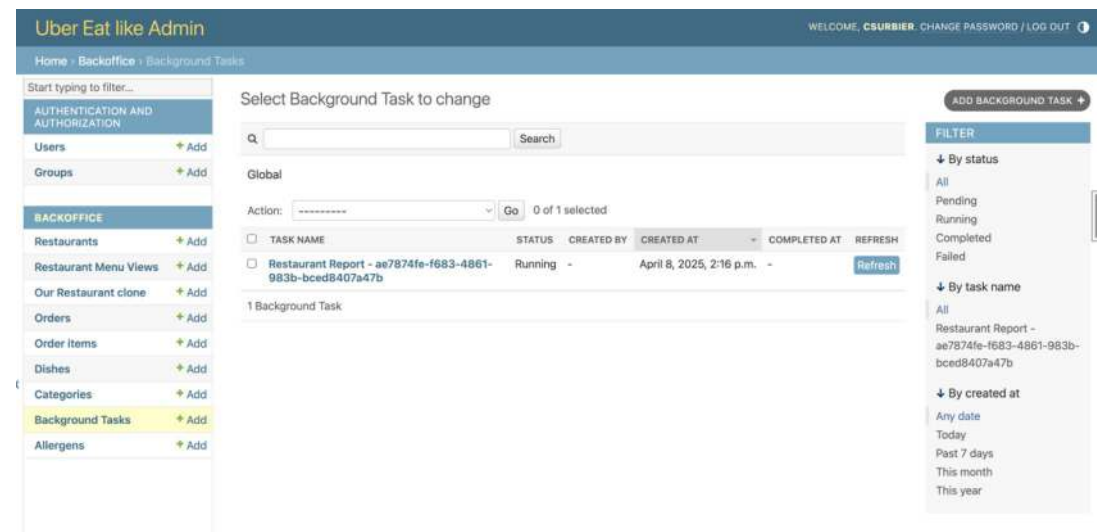


**Figure 13-4.** Admin-based task tracking list (*BackgroundTask* model) while jobs are running

And a few moments later, we can check that the status is completed as shown in Figure 13-5:

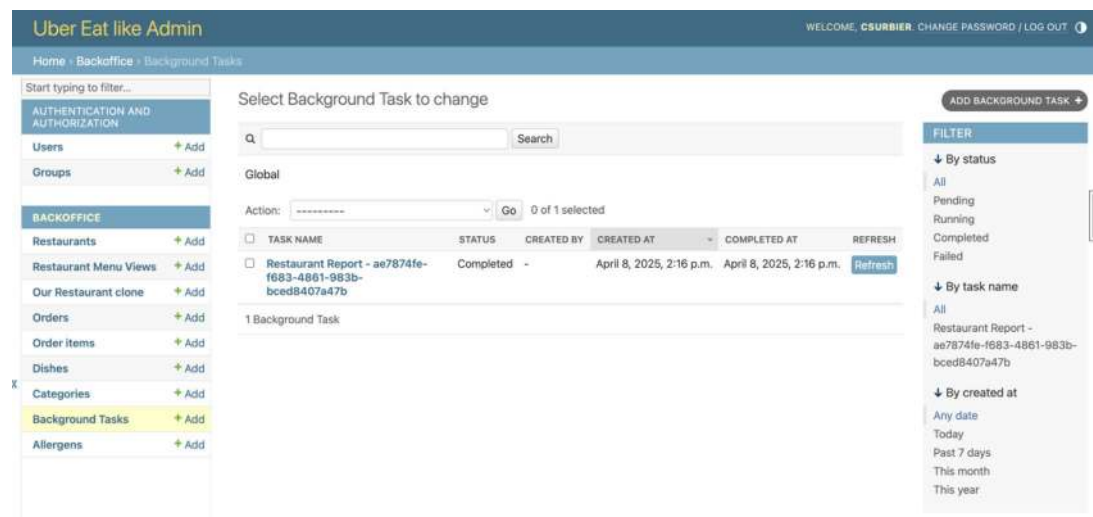


**Figure 13-5.** Task status updated after refreshing the admin task list

We could also have implemented custom URLs and views for better tasks management interface in the admin interface.

## CHAPTER 14

# Advanced Query Optimization

## Raw SQL Optimization When Needed

We learned to use **`select_related`** and **`prefetch_related`** to optimize our queries and how it's important to do it to keep optimal performances. However, while Django's ORM is powerful, sometimes raw SQL is necessary for complex optimizations. It may also be necessary to use raw SQL when Django's ORM is not able to generate the query we need (or we don't succeed to do what we want).

To write custom SQL, we need to import **`connection`** from **`django.db`**:

```
from django.db import connection
```

Let's add to our `RestaurantAdmin` class a method to get and display the five TOP dishes ordered by our customers (see Figure 14-1) using a raw SQL query:

```
class RestaurantAdmin(admin.ModelAdmin):
 # ... (previous code).
 @admin.display(description="Top dishes")
 def get_top_dishes(self, obj):
 """Get top dishes using raw SQL for better performance"""
 with connection.cursor() as cursor:
 cursor.execute("""
 SELECT d.name, COUNT(oi.id) as order_count
 FROM backoffice_dish d
 JOIN backoffice_orderitem oi ON d.id = oi.dish_id
 JOIN backoffice_order o ON oi.order_id = o.id
 WHERE d.restaurant_id = %s
 GROUP BY d.id
```



```
ORDER BY order_count DESC
LIMIT 5
"", [str(obj.id)])

return [f"{row[0]} ({row[1]} orders)" for row in cursor.fetchall()]
```

- 1. We use the **with connection.cursor() as cursor:** to create a cursor object that we will use to execute the SQL query.
- 2. We use the **cursor.execute()** method to execute the SQL query.
- 3. We use the **cursor.fetchall()** method to fetch all the results of the query.

And now we can add the method in the list of our displayed fields:

```
list_display = ('colored_name', 'toggle_active_button', 'total_orders',
'get_top_dishes', ...
```

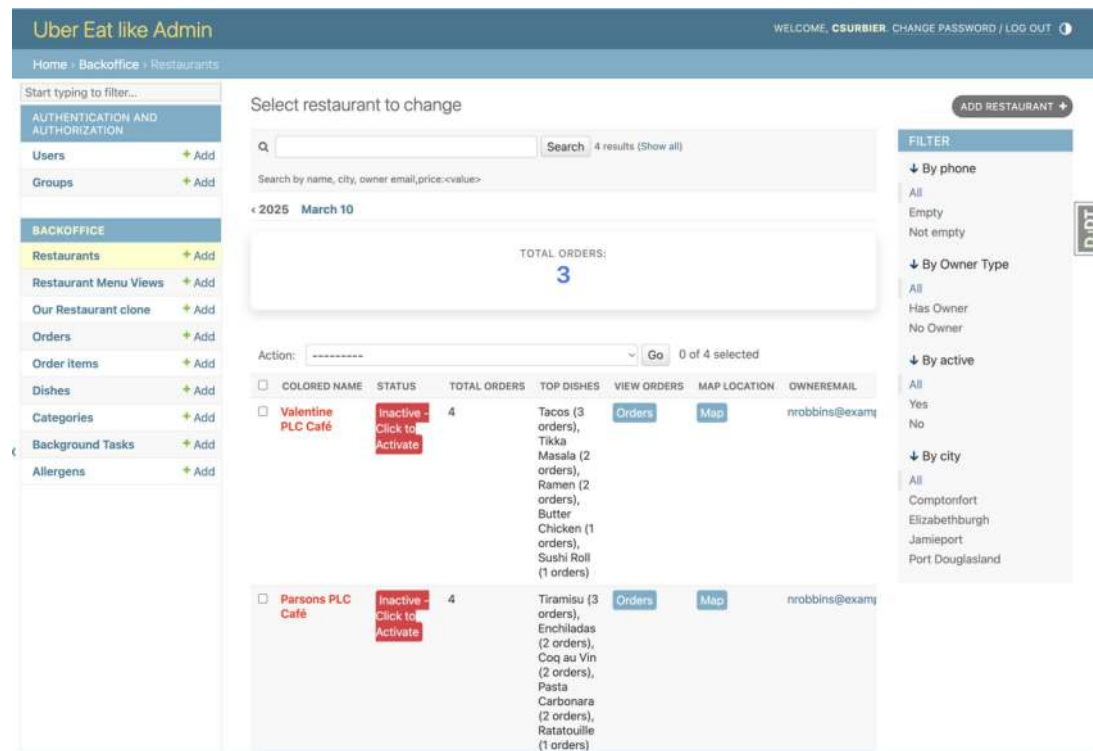


**Figure 14-1.** Raw SQL query results displayed in the admin as a custom list\_display field

---

**Note** This is a simple example of how to use raw SQL queries in Django Admin. For the example, we could have used the query

```
Dish.objects.filter(restaurant=obj).annotate(order_
count=Count('orderitem')).order_by('-order_count')[:5]
```

instead of the raw SQL query.

---

## Query Debugging and Profiling Tools

We already talked about the Django debug toolbar. Another useful tool is [Django Silk](https://django-silk.readthedocs.io/en/latest/) (<https://django-silk.readthedocs.io/en/latest/>). It provides a lot of useful information about the requests and queries made to the database (as shown on Figure 14-2 and Figure 14-3).

Let's install it:

```
pip install django-silk
```

Configure it in our **settings.py**:

```
INSTALLED_APPS = [
 'backoffice',
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'debug_toolbar',
 'django_admin_inline_paginator',
 'silk',
]

MIDDLEWARE = [
 'django.middleware.locale.LocaleMiddleware',
 'django.middleware.security.SecurityMiddleware',
 'django.contrib.sessions.middleware.SessionMiddleware',
 'django.middleware.common.CommonMiddleware',
```

```

'django.middleware.csrf.CsrfViewMiddleware',
'django.contrib.auth.middleware.AuthenticationMiddleware',
'django.contrib.messages.middleware.MessageMiddleware',
'django.middleware.clickjacking.XFrameOptionsMiddleware',
'debug_toolbar.middleware.DebugToolbarMiddleware',
'silk.middleware.SilkyMiddleware',
]

SILKY_PYTHON_PROFILER = True
SILKY_ANALYZE_QUERIES = True

```

Then we need to run migration to create **Silk**-related tables:

```
python manage.py migrate
python manage.py collectstatic
```

and add it to our **urls.py**:

```
urlpatterns = [
 path('admin/', admin.site.urls),
 path('api/categories/', views.get_categories, name='get_categories'),
 path('api/get-category-dishes/', views.get_category_dishes, name='get-
category-dishes'),
 path('silk/', include('silk.urls', namespace='silk')),
]
```

Now, we can start browsing the admin interface and then go to our Silk dashboard to get information about requests and queries made.

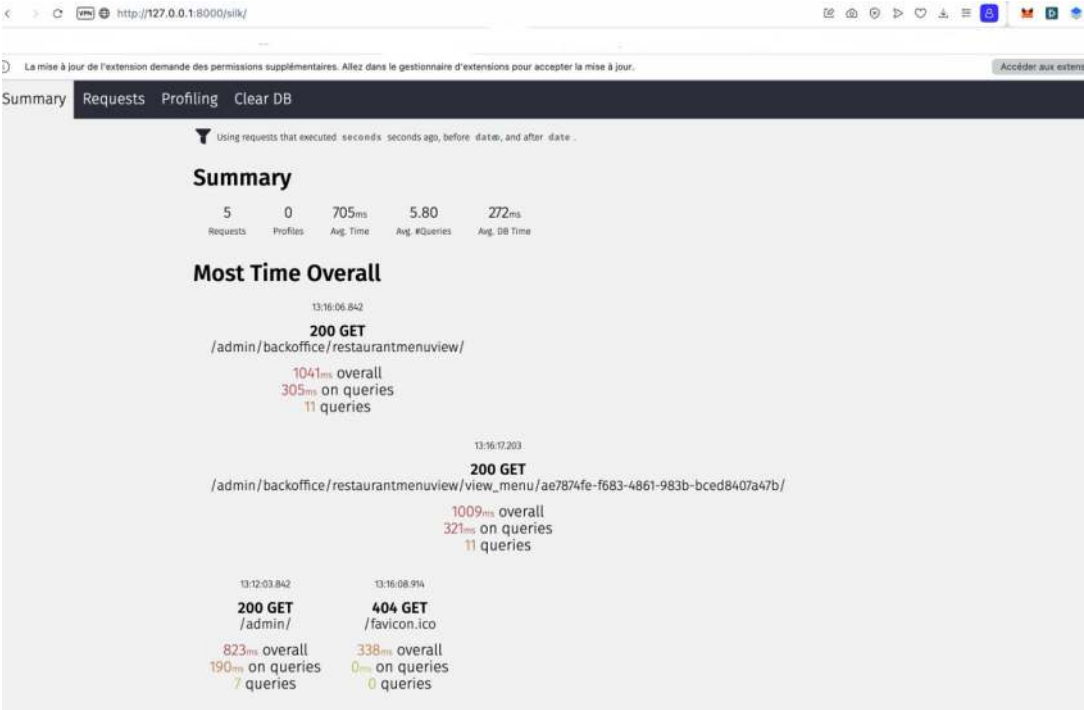


**Figure 14-2.** Django Silk dashboard for request/query profiling

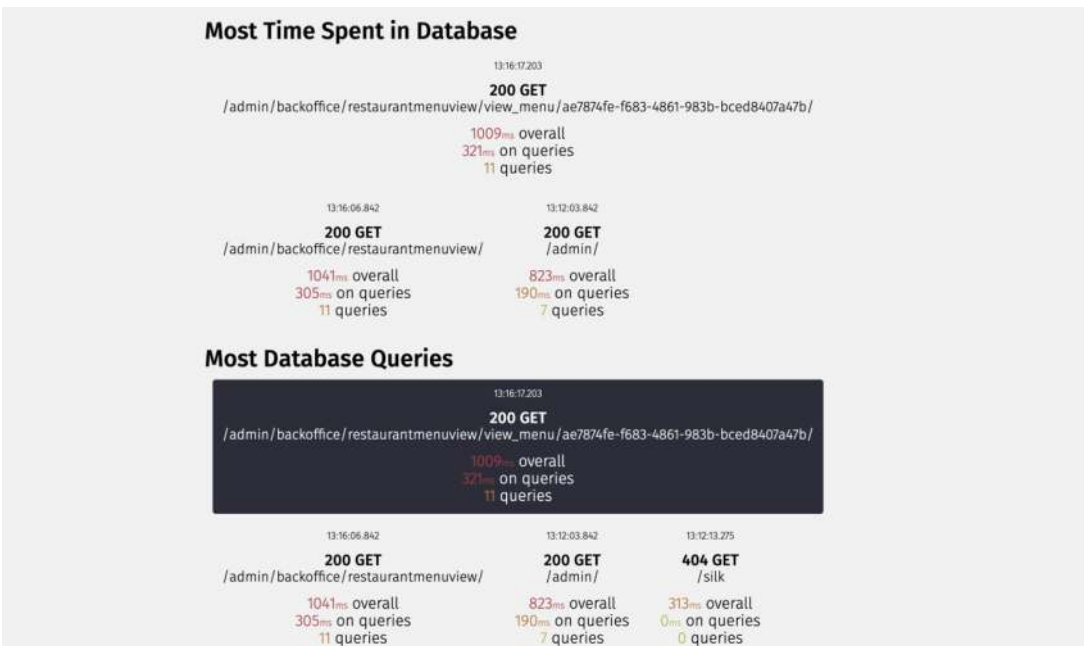


**Figure 14-3.** Django Silk detailed profiling view (requests and SQL queries)

## Summary

In this chapter, we've started to explore advanced customization techniques that transform Django Admin from a basic CRUD interface into a powerful, tailored administrative tool.

We began by enhancing list displays with custom links and actions, allowing administrators to navigate efficiently between related data and perform operations directly from the list view. We then explored how to implement chained filters and integrate database views, providing intuitive ways to navigate complex data relationships.

The deep dive into Django's template system revealed how to completely transform the admin interface by overriding templates, customizing forms, and implementing Ajax functionality for dynamic interactions. We learned how to create custom views and URLs within the admin interface, enabling specialized workflows and dashboards.

Our exploration of REST API integration demonstrated how to connect the admin interface with external systems, while our implementation of background tasks with Celery showed how to handle resource-intensive operations without blocking the user interface.

Finally, we examined how to use raw SQL queries to ensure our admin interface remains performant even when dealing with large datasets and complex relationships.

By mastering these advanced customization techniques, you should now have an overview of how to transform Django Admin into a more tailored solution to match your application's administrative needs. Django Admin can be customized to provide an intuitive and efficient interface for your team.

In the next chapter, we will continue to explore advanced customization techniques to further enhance the Django Admin interface and rebrand it to match the look and feel that you want!

# PART IV

## UI/UX Enhancement

This part focuses on elevating the admin's look and feel, from deep template customization and modern CSS frameworks to responsive design, custom dashboards, and interactive charts.

## CHAPTER 15

# Template System Deep Dive

We learned in the previous chapter that each Django Admin template can be overridden. Let's dive deeper into the template system.

## Block Override Patterns

The Django Admin interface uses a powerful template inheritance system with block tags that allow customization without duplicating code.

## Template Inheritance Hierarchy

Django Admin templates follow this inheritance chain:

- ``admin/base.html`` (root template)
  - ``admin/base_site.html`` (extends `base.html`)
    - ``admin/index.html`` (extends `base_site.html`)
    - ``admin/change_list.html`` (extends `base_site.html`)
    - ``admin/change_form.html`` (extends `base_site.html`)

Let's explore some main key block tags we could find in admin templates.

## In Template `base.html`

The **`base.html`** template serves as the foundation for all admin templates. It defines the basic structure and layout of the admin interface.

Common Block Tags in **base.html** template: ##### {% block title %}:

```
{% block title %}
 {{ title }} | {{ site_title|default:_('Django site admin') }}
{% endblock %}
```

This block defines the page title that appears in the browser tab. When customizing, you can override this to provide a more descriptive title for your admin dashboard.

## {% block stylesheet %}

This block is used to include default Django Admin stylesheets.

## {% block extrahead %}

We can use this block to include any additional JavaScript code or meta tags in the head section of the page.

## {% block extrastyle %}

```
{% block extrastyle %}
 {{ block.super }}
 <link rel="stylesheet" type="text/css" href="{% static 'admin/css/
 dashboard.css' %}">
{% endblock %}
```

This block allows you to add additional CSS stylesheets. The **{{ block.super }}** variable includes the content from the parent template's block, so your styles are added to the existing ones rather than replacing them.

---

When we want to add additional item, rather replacing the default block, we need to use the **{{ block.super }}** variable.

---

## {% block nav-global %}

This block is used to include any global navigation elements.



**{% block nav-sidebar %}**

This block is used to include the navigation sidebar.

**{% block messages %}**

This block is used to display messages, such as success or error messages.

**{% block usertools %}**

This block is used to include the user tools, such as the user's name and logout link.

**{% block footer %}**

This block is used to include the footer content.

**In template index.html****{% block coltype %}**

```
{% block coltype %}colMS{% endblock %}
```

This defines the column type for the layout. In the admin index, colMS indicates a main section with a sidebar.

**{% block bodyclass %}**

```
{% block bodyclass %}{{ block.super }} dashboard{% endblock %}
```

This adds CSS classes to the body element. The dashboard class is specific to the index page.

**{% block nav-breadcrumbs %}**

```
{% block nav-breadcrumbs %}{% endblock %}
```

This block is used to display the navigation breadcrumbs.

## {% block content %}

```
{% block content %}
<div id="content-main">
 <!-- Dashboard stats -->
 <div class="dashboard-stats">
 <!-- Stat cards -->
 </div>

 <!-- App list -->
 {% include "admin/app_list.html" with app_list=app_list show_
 changelinks=True %}
</div>
{% endblock %}
```

This is the main content area of the page. In the index template, it typically contains the dashboard statistics and the app list.

## {% block sidebar %}

```
{% block sidebar %}
<div id="content-related">
 <div class="module" id="recent-actions-module">
 <h2>{% trans 'Recent actions' %}</h2>
 <h3>{% trans 'My actions' %}</h3>
 {% load log %}
 {% get_admin_log 10 as admin_log for_user user %}
 <!-- Recent actions list -->
 </div>
</div>
{% endblock %}
```

This block defines the sidebar content, which typically shows recent actions in the admin.

It will be too long to review in this book all block tags in admin templates. But now we have an overview of the most common ones. When we want to override a template, we can have a look at the original template to see which blocks are available.

## How Block Tags Work in Template Inheritance

When Django renders a template that extends another template (like `{% extends "admin/base_site.html" %}`), it follows these steps:

- It loads the parent template (`base_site.html`).
- For each block in the parent, it checks if the child template overrides that block.
- If the block is overridden, it uses the child's content; otherwise, it uses the parent's content.

This inheritance system allows us to

- Override specific sections of a template without duplicating the entire template
- Reuse the parent template's structure while customizing specific parts
- Use `{{ block.super }}` to include the parent's content and add to it rather than replacing it

**Example:** Customizing the content block: Here's how we have customized the content block to add our dashboard statistics:

```
{% extends "admin/base_site.html" %}
{% load i18n static %}

{% block extrastyle %}
 {{ block.super }}
<style>
 /* Your custom CSS for dashboard stats */
```

```

</style>
{% endblock %}

{% block content %}
<div id="content-main">
 <!-- Custom Dashboard Stats -->
 <div class="dashboard-stats">
 <div class="stat-card">
 <h3>{{ total_restaurants }}</h3>
 <p>Total Restaurants</p>
 </div>
 <!-- More stat cards -->
 </div>

 <!-- Keep the standard app list -->
 {% include "admin/app_list.html" with app_list=app_list show_
 changelinks=True %}
</div>
{% endblock %}

```

## Template Resolution Order

Django searches for admin templates in this order:

1. `our_app/templates/admin/our_app/your_model/template_name.html`
2. `our_app/templates/admin/our_app/template_name.html`
3. `our_app/templates/admin/template_name.html`
4. `django/contrib/admin/templates/admin/template_name.html`

### Example:

For Restaurant `change_form.html`:

1. `backoffice/templates/admin/backoffice/restaurant/change_form.html` (most specific)
2. `backoffice/templates/admin/backoffice/change_form.html` (app-level)

3. `backoffice/templates/admin/change_form.html` (global admin override)
4. Django's default (fallback)

## Best Practices for Working with Block Tags

- **Use `{{ block.super }}` when appropriate:** This allows us to extend rather than replace the parent block's content.
- **Keep block overrides focused:** Override only the blocks we need to change, letting Django use the parent template's defaults for the rest.

Understand the block hierarchy: Some blocks may be nested within others, so be aware of the structure when overriding.

Maintain consistent structure: When overriding blocks, try to maintain a similar structure to the original to ensure compatibility with Django's CSS and JavaScript.

By understanding and effectively using block tags in our Django Admin templates, we can create customized, feature-rich admin interfaces while leveraging Django's built-in functionality.

## Custom Tags

Django allows us to extend the template language by creating custom template tags and filters. This is particularly useful when we need to perform complex operations that aren't easily handled by the built-in tags.

## Why Use Custom Tags in Django Admin?

Custom template tags become essential when we need to

- Calculate statistics or aggregations for dashboard widgets
- Format data in ways not covered by built-in filters
- Create reusable components across multiple admin templates
- Access model methods or properties that require parameters

## Creating Custom Template Tags

To create custom template tags for our Django Admin, we need to

1. Create a `templatetags` directory in your app
2. Add an `__init__.py` file to make it a Python package
3. Create a Python file for our tags (e.g., `admin_extras.py`)

To illustrate how to create and use custom tags, we will modify our **custom\_index** method in our **backoffice/admin.py** file, to comment the **extra\_context** variable that we added in the previous chapter:

```
def custom_index(request, extra_context=None):
 # Initialize extra_context if it's None
 extra_context = extra_context or {}
 from django.db.models import Count
 # Add custom statistics to the context
 """
 extra_context.update({
 'total_restaurants': Restaurant.objects.count(),
 'active_restaurants': Restaurant.objects.filter(is_active=True).
 count(),
 'total_dishes': Dish.objects.count(),
 'recent_orders': Order.objects.filter(created_at__gte=datetime.
 now() - timedelta(days=7)).count()
 })
 """
 # Call the original index method correctly
 return admin.site.__class__.index(admin.site, request, extra_context)
```

and then we will create custom tags in the **backoffice/templatetags/admin\_extras.py** file:

```
from datetime import datetime, timedelta
from django import template
from django.db.models import Count, Avg
```

```

from django.utils.safestring import mark_safe
from backoffice.models import Restaurant, Dish, Order

register = template.Library()

@register.simple_tag
def total_restaurants():
 """Return the total number of restaurants"""
 return Restaurant.objects.count()

@register.inclusion_tag('admin/dashboard_stats.html')
def dashboard_statistics():
 """Render dashboard statistics widget"""
 context = {
 'active_restaurants': Restaurant.objects.filter(is_active=True).
 count(),
 'total_dishes': Dish.objects.count(),
 'recent_orders': Order.objects.filter(created_at__gte=datetime.
 now() - timedelta(days=7)).count()
 }
 return context

```

1. We use a simple tag to display a count of total restaurants.
2. We use an inclusion tag to display a more complex widget with multiple statistics:
  1. We need to create a template file to render the widget (**backoffice/templates/admin/dashboard\_stats.html**).
  2. We return a dictionary in the context of the inclusion tag that contains the data we want to display in the widget.

In the **backoffice/templates/admin/dashboard\_stats.html** file, we will add the following code:

```

<div class="stat-card">
 <h3>{{ active_restaurants }}</h3>
 <p>Active Restaurants from our custom dashboard stats</p>
</div>

```

```

<div class="stat-card">
 <h3>{{ total_dishes }}</h3>
 <p>Total dishes from our custom dashboard stats</p>
</div>
<div class="stat-card">
 <h3>{{ recent_orders }}</h3>
 <p>Total recent orders from our custom dashboard stats</p>
</div>

```

## Using Custom Tags in Admin Templates

To use our custom tags in our custom admin index template, we will edit our **backoffice/templates/admin/index.html** file and just need to use syntax `{% %}` to use tags in the template:

```

{% extends "admin/base_site.html" %}
{% load i18n static admin_extras %}

{% block extrastyle %}
 {{ block.super }}
<style>
 .dashboard-stats {
 display: grid;
 grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
 gap: 20px;
 margin-bottom: 30px;
 }
 .stat-card {
 background: #fff;
 border: 1px solid #ddd;
 border-radius: 8px;
 padding: 20px;
 text-align: center;
 box-shadow: 0 2px 4px rgba(0,0,0,0.1);
 }
 .stat-number {

```



```

 font-size: 2.5em;
 font-weight: bold;
 color: #417690;
 }
</style>
{% endblock %}

{% block content %}
<div id="content-main">
 <div class="dashboard-stats" >
 <div class="stat-card">
 <h3>{% total_restaurants %}</h3>
 <p>Total Restaurants</p>
 </div>
 <!-- Using inclusion tag for complex widgets -->
 {% dashboard_statistics %}
 </div>
 {% include "admin/app_list.html" with app_list=app_list show_
changelinks=True %}
</div>
{% endblock %}

```

In our **{% block content %}** block, we used our new custom tags to display the total number of restaurants and the statistics widget as shown on Figure 15-1.

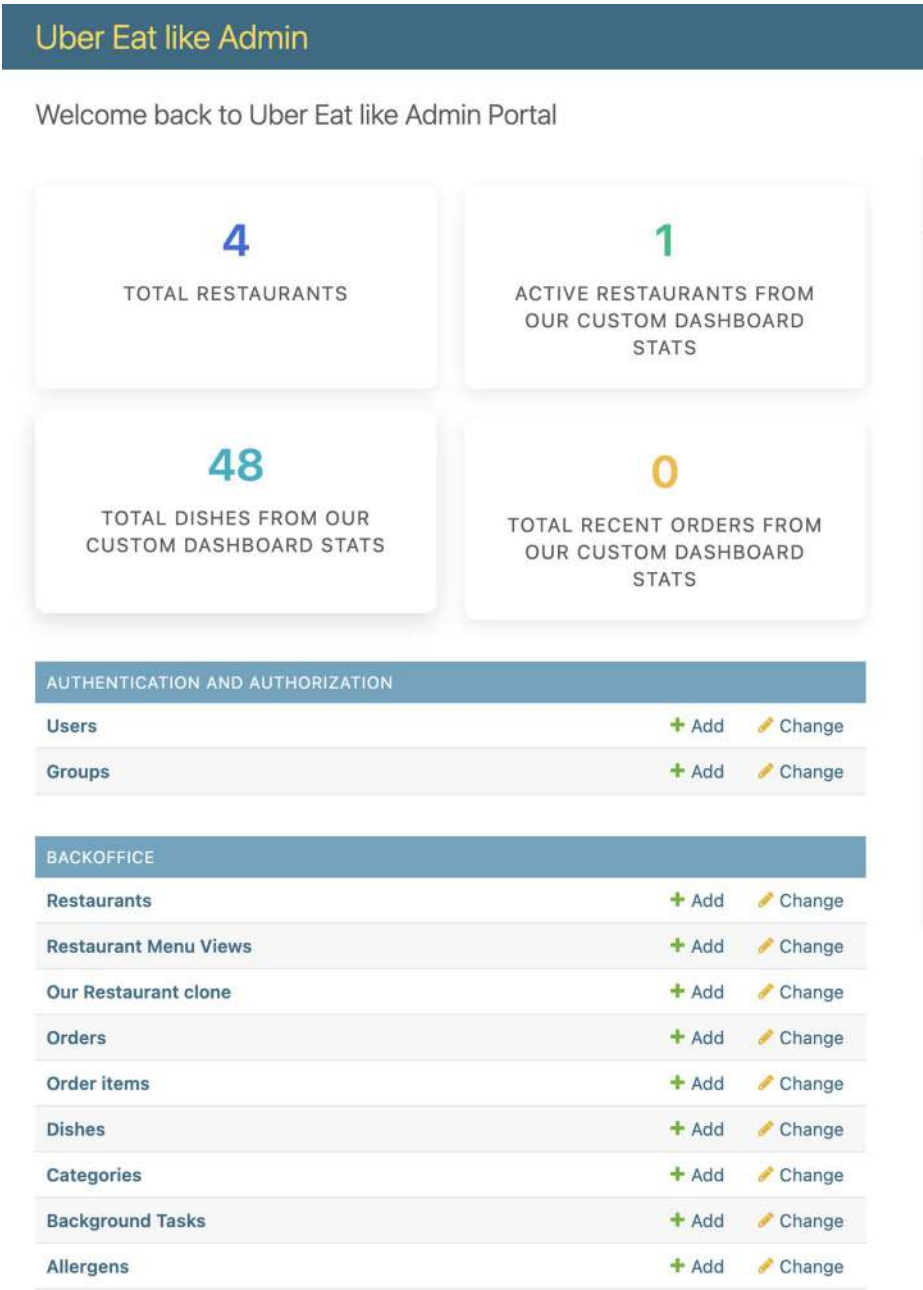


**Figure 15-1.** Admin index using custom template tags (restaurant count and statistics widget)

## Best Practices for Custom Tags

1. **Keep tags focused:** Each tag should have a single, clear purpose.
2. **Handle errors gracefully:** Always include try/catch blocks for database operations.
3. **Use caching when appropriate:** For expensive operations, consider Django's cache framework.
4. **Document your tags:** Include docstrings explaining what each tag does.
5. **Test your tags:** Write unit tests for complex custom tags.

```
from django.core.cache import cache

@register.simple_tag
def cached_statistics():
 """Get dashboard statistics with caching"""
 cache_key = 'dashboard_stats'
 stats = cache.get(cache_key)

 if stats is None:
 stats = {
 'active_restaurants': Restaurant.objects.filter(is_active=True).
 count(),
 'total_dishes': Dish.objects.count(),
 'recent_orders': Order.objects.filter(created_at__gte=datetime.
 now() - timedelta(days=7)).count()
 }
 # Cache for 5 minutes
 cache.set(cache_key, stats, 300)

 return stats
```

## Custom Filters

Custom filters provide a way to transform and format data directly in templates. Unlike tags, filters take an input value and return a modified output, making them perfect for data formatting and presentation enhancements in the Django Admin.

## Creating Custom Filters

Custom filters are simpler than tags and are ideal for data transformation. We will create custom filters to use with our **Dish** model.

```
@register.filter
def currency_format(value):
 """Format price values as currency"""
 if value is None:
 return "N/A"
 return f"${value:.2f}"

@register.filter
def truncate_description(text, length=50):
 """Truncate dish descriptions for list views"""
 if len(text) <= length:
 return text
 return f"{text[:length]}..."

@register.filter
def boolean_icon(value):
 """Display boolean values as colored symbols"""
 if value:
 return mark_safe('✓')
 else:
 return mark_safe('✗')
```

To use these filters, the syntax is straightforward. We include the filter name in the template, followed by the variable or value we want to apply the filter to, like this: **{{ variable|filter\_name }}**.

---

**Note** This example is only for demonstration purposes. Django already provides a built-in filter to truncate strings to a given length: <https://docs.djangoproject.com/en/6.0/ref/templates/builtins/#truncatechars>.

---

## Using Custom Filters in Templates

Now, we will create a custom **change\_list** template for the **Dish** model to use these filters (see Figure 15-2). We will just replace the `{% block result_list %}` block with the following code:

```
{% extends "admin/base_site.html" %}
{% load i18n admin_urls static admin_list admin_extras %}

...

{% block result_list %}
 {% if action_form and actions_on_top and cl.show_admin_actions %}
 {% admin_actions %}{% endif %}
 <div class="results">
 <table id="result_list">
 <thead>
 <tr>
 <th>Name</th>
 <th>Category</th>
 <th>Description</th>
 <th>Restaurant</th>
 <th>Price</th>
 <th>Available</th>
 </tr>
 </thead>
```

```

 <tbody>
 {% for dish in cl.result_list %}
 <tr class="{% cycle 'row1' 'row2' %}">
 <td>{{ dish.name }}</td>
 <td>{{ dish.category }}</td>
 <td>{{ dish.description|truncate_description:30 }}</td>
 <td>{{ dish.restaurant }}</td>
 <td>{{ dish.price|currency_format }}</td>
 <td>{{ dish.is_available|boolean_icon }}</td>
 </tr>
 {% endfor %}
 </tbody>
 </table>
</div>
{% if action_form and actions_on_bottom and cl.show_admin_actions
 %}{% admin_actions %}{% endif %}
{% endblock %}

...

```

1. We create a custom HTML table to display the dishes.
2. We use the custom filter **{{ dish.description|truncate\_description:30 }}** to truncate the description to 30 characters.
3. We use the custom filter **{{ dish.price|currency\_format }}** to format the price with currency symbol.
4. We use the custom filter **{{ dish.is\_available|boolean\_icon }}** to display a boolean value as a colored symbol.

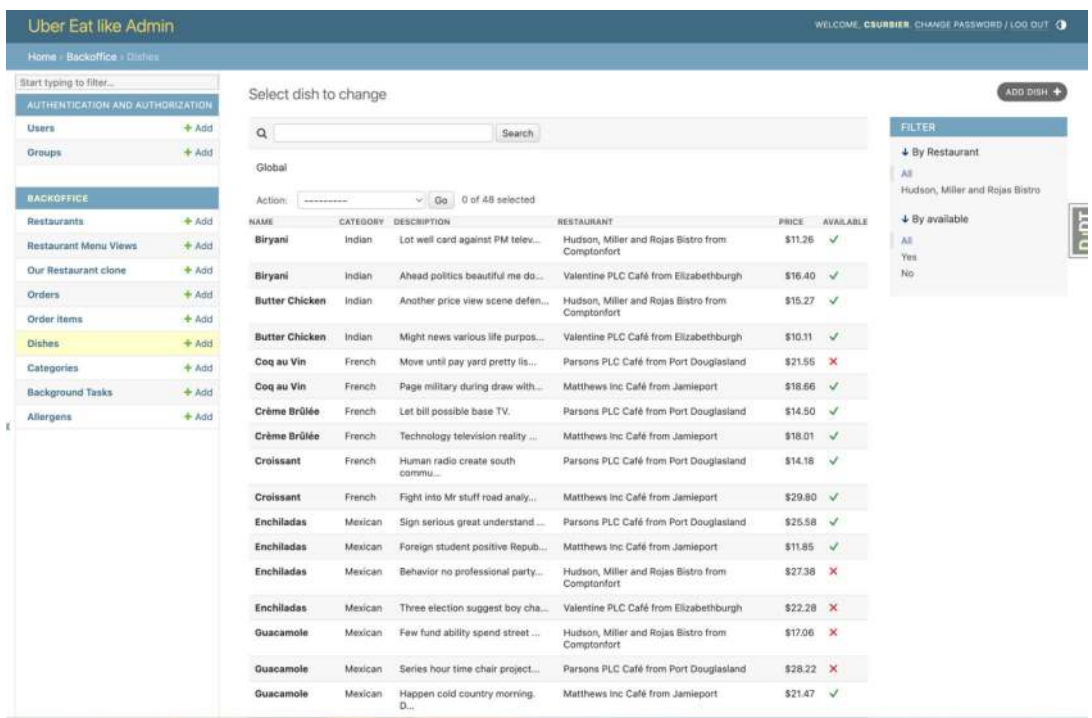


**Figure 15-2.** Dish changelist using custom template filters (currency, truncation, boolean icons)

Custom filters are particularly useful for consistent formatting across our admin interface, making data more readable and professional-looking while keeping the formatting logic separate from our models and views.

By leveraging custom template tags and filters, we can create a more dynamic and informative Django Admin interface that provides real-time insights and enhanced functionality while maintaining the clean separation between logic and presentation that Django promotes.

# Existing Packages to Rebrand Django Admin

We learned how to customize **Django Admin templates**, and with a lot of perseverance and time, we could rebrand the full **Django Admin** look and feel interface, **or** we could use some existing packages which do the job for us.

Let’s review some existing packages to rebrand Django Admin.

## Django Jet

One of my favorite was **Django jet** (<https://github.com/geex-arts/django-jet>), but it’s not maintained anymore. For new Django versions 4 and 5, we could use the new **Django jet reboot** (<https://github.com/assem-ch/django-jet-reboot>) package.

And some screenshots are shown (Figures 15-3 and 15-4).

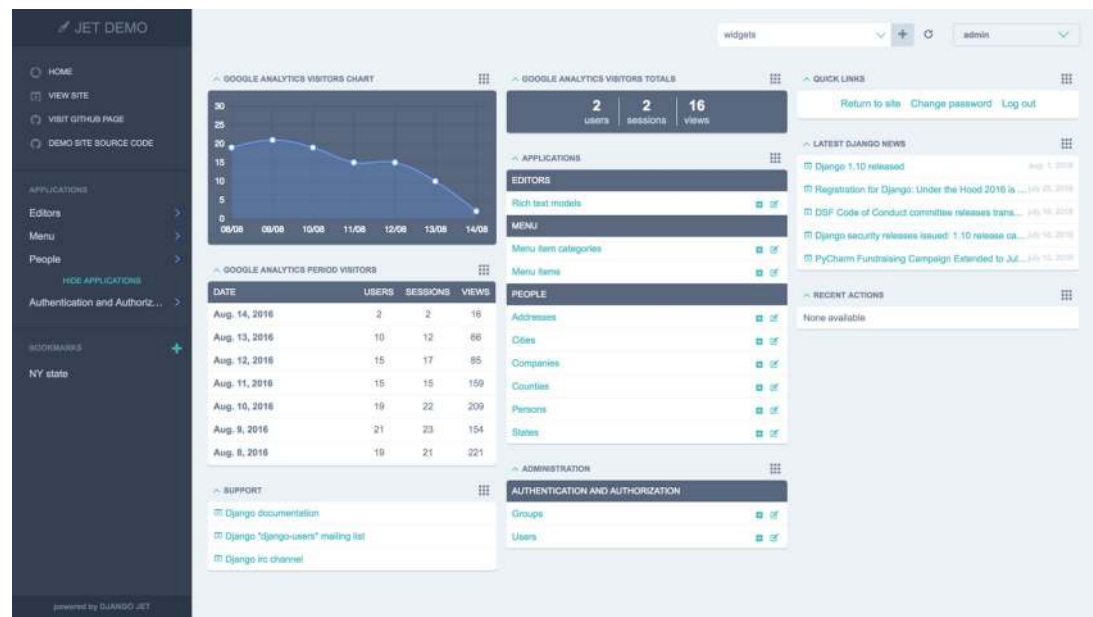


Figure 15-3. Django Jet admin interface (screenshot 1)



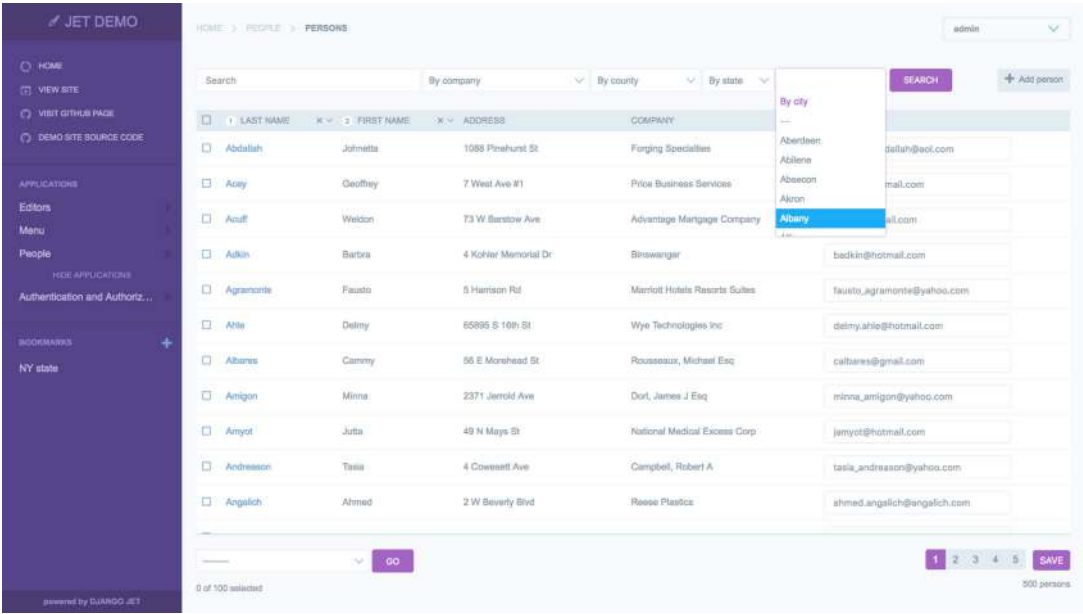


**Figure 15-4.** Django Jet admin interface (screenshot 2)

# Django Grapelli

Django Grapelli (<https://grappelliproject.com/>) is another package (see Figure 15-5) still active

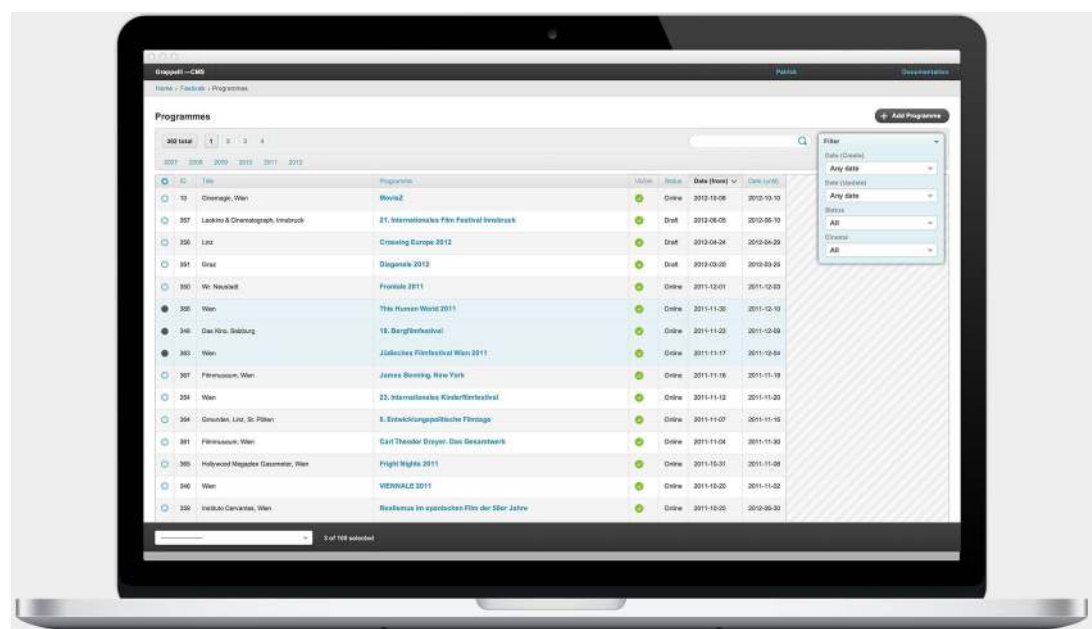


Figure 15-5. Django Grappelli rebranded admin appearance

# Django Jazzmin

Django jazzmin (<https://github.com/faridav/django-jazzmin>) which comes with different themes ([https://django-jazzmin.readthedocs.io/ui\\_customisation/](https://django-jazzmin.readthedocs.io/ui_customisation/)) as shown in Figure 15-6 and Figure 15-7.

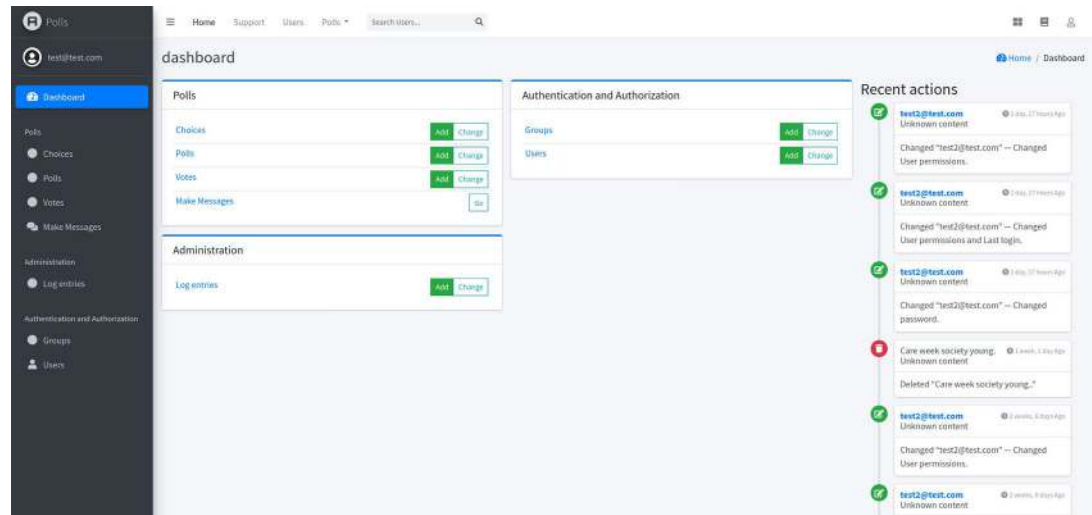


Figure 15-6. Django Jazzmin dashboard theme (default)

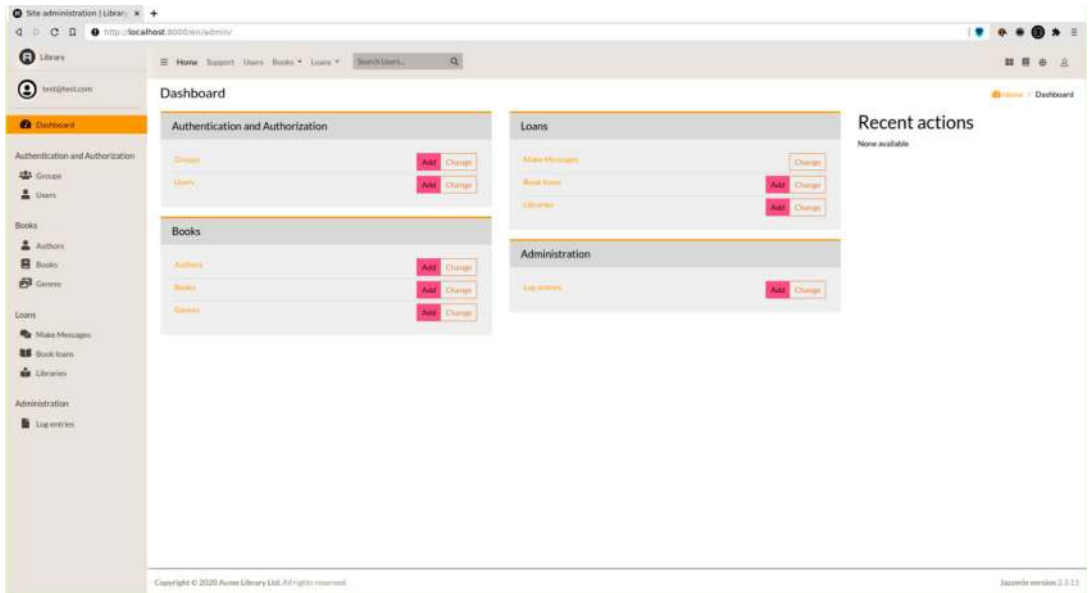


**Figure 15-7.** Django Jazzmin with Darkly theme variant

## Django Daisy

Django daisy (<https://github.com/hypy13/django-daisy>) is a modern, sleek, and highly responsive admin dashboard built with DaisyUI and TailwindCSS (see Figure 15-8 and Figure 15-9).

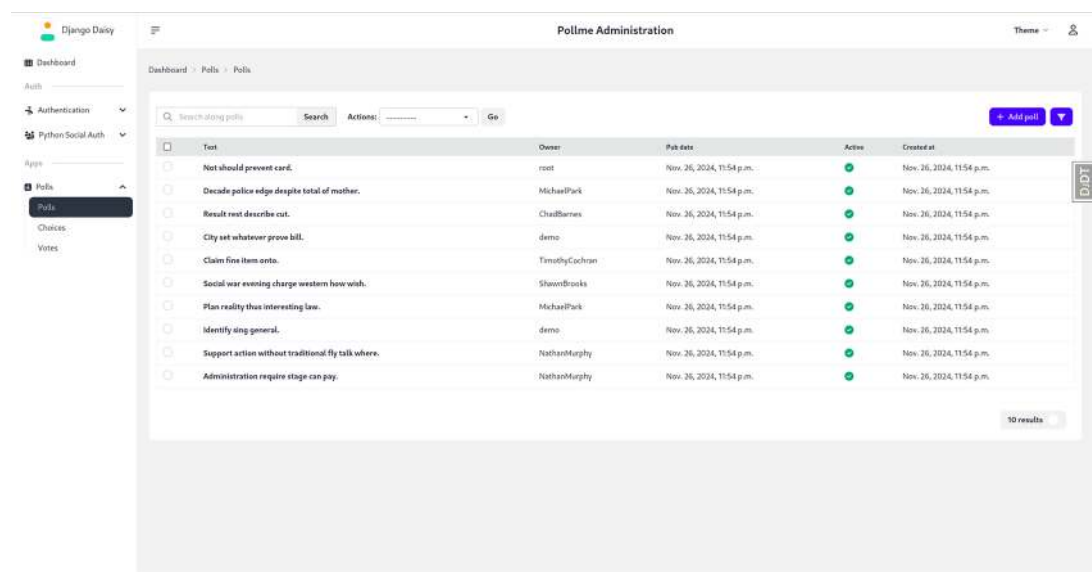


Figure 15-8. Django Daisy admin list view (DaisyUI / Tailwind)

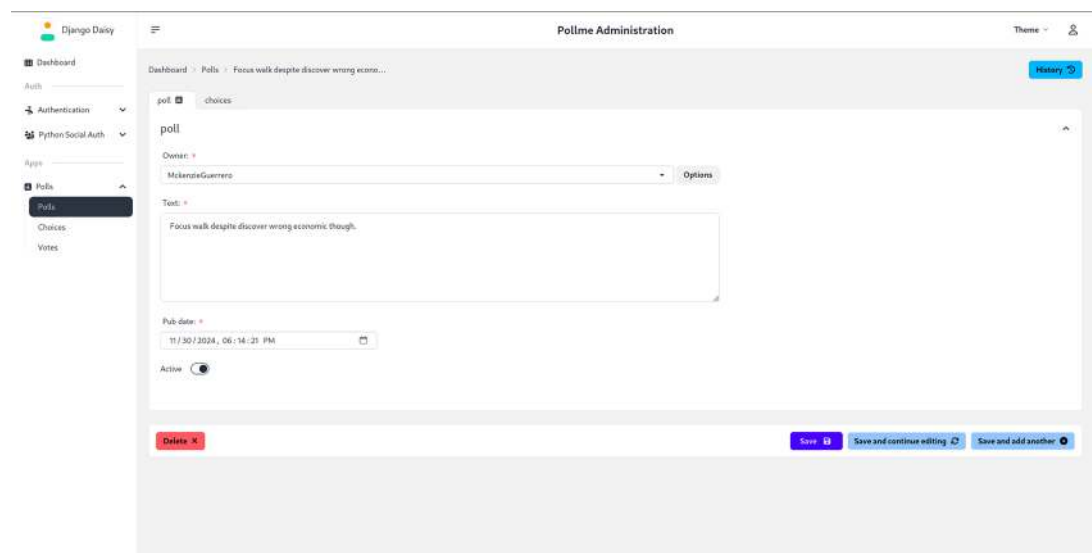


Figure 15-9. Django Daisy change form view

# Django Unfold

Django unfold (<https://unfoldadmin.com/>) is a new player in the Django Admin interface customization tools. It’s a powerful UI framework designed to make the

Django Admin more modern, user-friendly, and visually appealing (see Figure 15-10, 15-11, 15-12). **Django-Unfold** brings a modern, clean, and well-structured design while maintaining Django’s powerful admin functionalities.

Let’s see how to use it. First, we need to install the package:

```
pip install django-unfold
```

Then, in the `INSTALLED_APPS` section of our **settings.py**, we need to add the “**unfold**” instruction:

```
INSTALLED_APPS = [
 "unfold", # before django.contrib.admin
 'backoffice',
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'debug_toolbar',
 'silk',
 'django_admin_inline_paginator',
]
```

Now, we need to run the **collectstatic** command, to install the **unfold** files:

```
python manage.py collectstatic
```

and then edit our **admin.py** file:

1. Import **from unfold.admin import ModelAdmin**.
2. Replace all **admin.ModelAdmin** by **ModelAdmin**.

```
from unfold.admin import ModelAdmin
class BaseModelAdmin(ModelAdmin):
 ...

@admin.register(Restaurant)
class RestaurantAdmin(ModelAdmin):
 ...
```

And only with that minimal configuration, we already have our new admin interface.

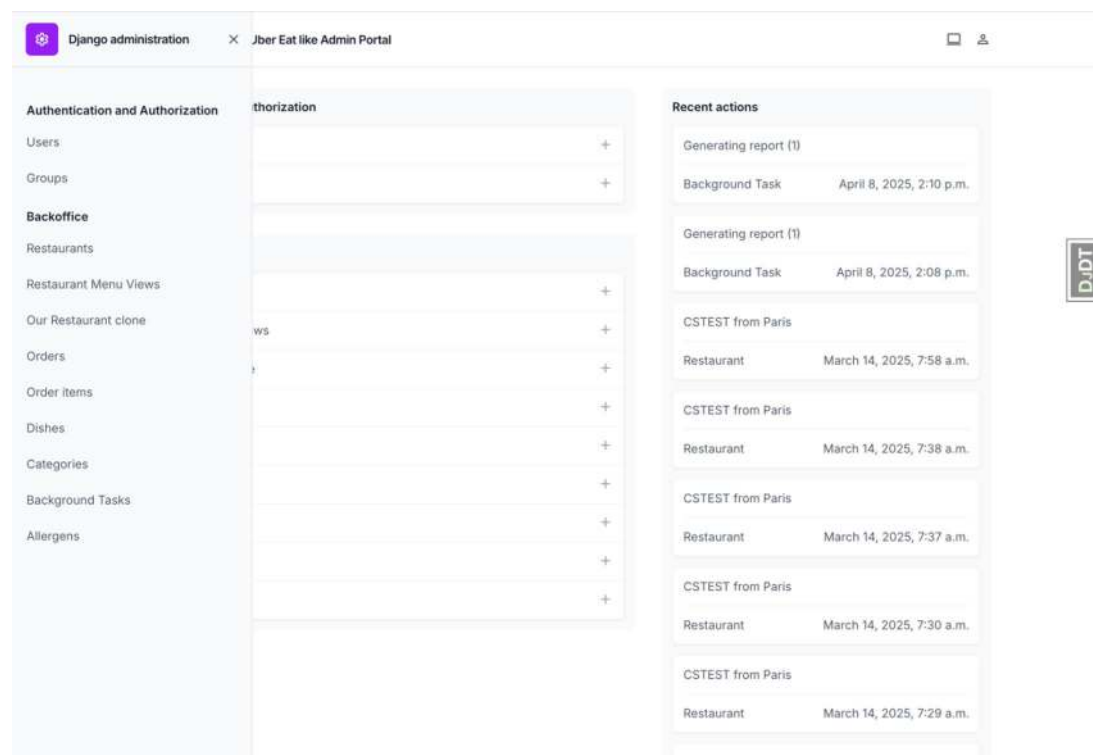


**Figure 15-10.** *Django Unfold: main admin index after minimal configuration*

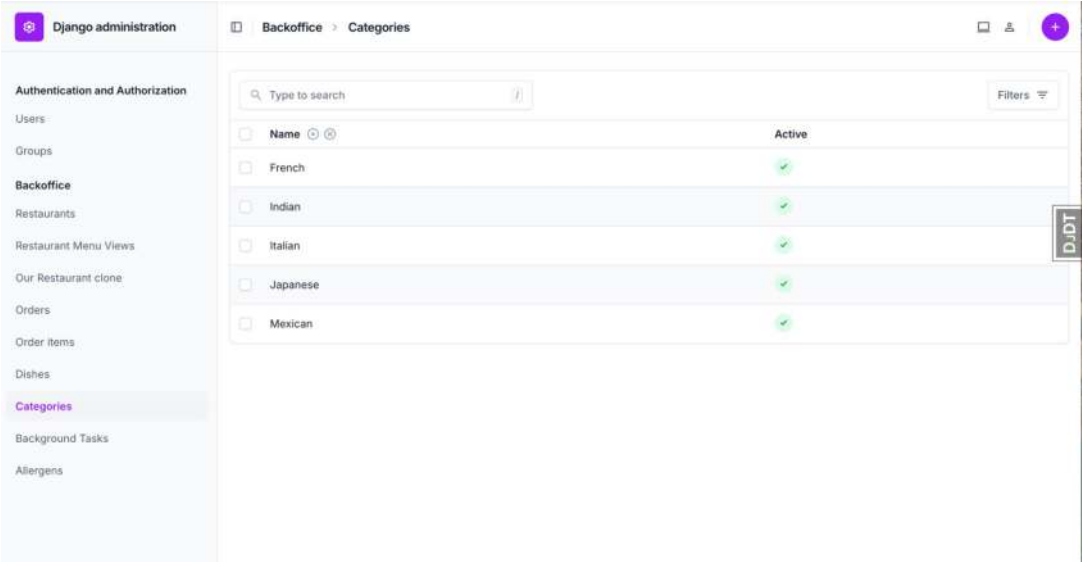


**Figure 15-11.** Django Unfold: categories changelist

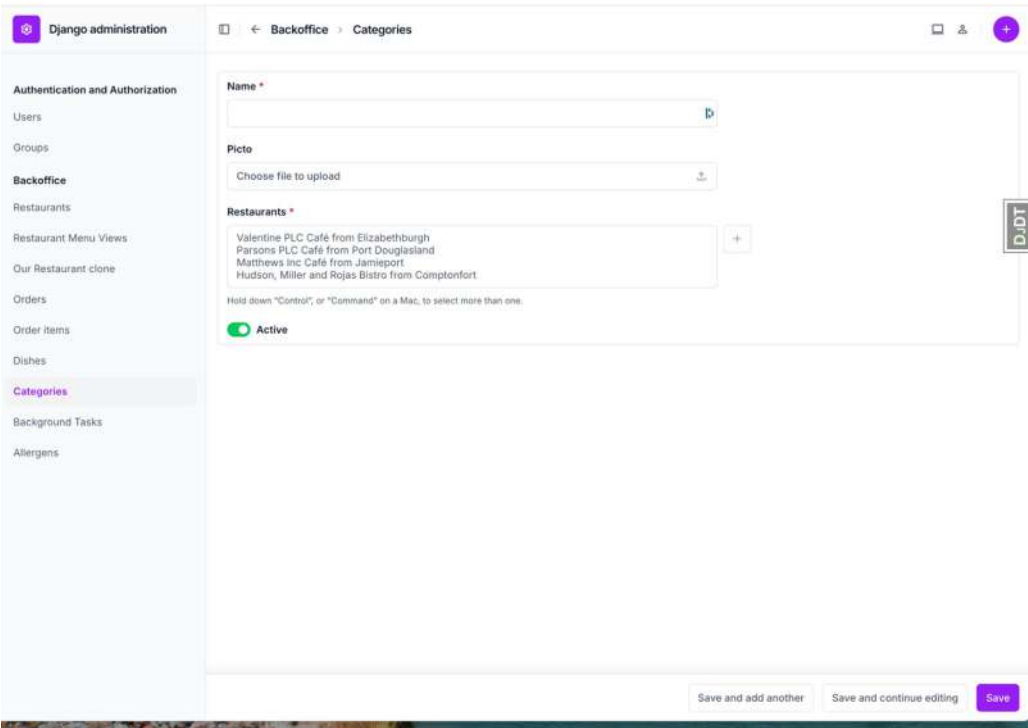


**Figure 15-12.** Django Unfold: add category form

To learn more about **Unfold** and its amazing possibilities, please refer to the [Documentation](https://unfoldadmin.com/docs/installation/quickstart/) (<https://unfoldadmin.com/docs/installation/quickstart/>) and don't hesitate to have a look at their [Demo](https://demo.unfoldadmin.com/en/admin/) (<https://demo.unfoldadmin.com/en/admin/>) and its [source code](https://github.com/unfoldadmin/formula) (<https://github.com/unfoldadmin/formula>).



## CHAPTER 16

# Modern CSS Frameworks

We learned about the **Django Admin** template inheritance and reviewed some existing packages to rebrand the interface, but if you still want to create your own look and feel, in this chapter, we will explore how to integrate popular CSS frameworks to create a more polished, responsive, and visually appealing admin experience.

---

Please remove all previous installed Django Admin package to continue with that chapter.

---

## Bootstrap Integration

Let's start with Bootstrap integration. Bootstrap remains one of the most popular CSS frameworks, offering a comprehensive component library and responsive grid system.

There are different ways to integrate Bootstrap in our admin, and the first one is to use the **cdn version** of it.

## Bootstrap cdn Version

Let's override the default **base\_site.html** template:

```
cp ../venv/lib/python3.12/site-packages/django/contrib/admin/templates/
admin/base_site.html backoffice/templates/admin/
```

And then we will modify it to include **Bootstrap**:

```
<!-- templates/admin/base_site.html -->
{% extends "admin/base.html" %}
{% load static %}
```

```
{% block title %}{% title %} | {% site_title|default:_('Django site admin')
%}{% endblock %}

{% block extrahead %}
 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/
 bootstrap.min.css" rel="stylesheet">
{% endblock %}

{% block extrajs %}
 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/
 bootstrap.bundle.min.js"></script>
{% endblock %}

{% block branding %}
<h1 id="site-name">
 Uber Eat like
 Bootstrap Admin
</h1>
{% endblock %}
```

Now let's modify our **admin.py** to include a specific Bootstrap instruction to check that everything is working and that we can use **Bootstrap**. We will modify our **view\_orders** method from our **RestaurantAdmin** classes, to use an alert box and a Bootstrap button ( see Figure 16-1):

```
@admin.register(Restaurant)
class RestaurantAdmin(admin.ModelAdmin):
 ...

@admin.display(description="View Orders")
def view_orders(self, obj):
 return format_html(
 '<div class="alert alert-primary" role="alert">Click below to see
 the orders for this restaurant</div><a class="btn btn-light btn-
 sm" href="/admin/backoffice/order/?restaurant__id__exact={}"
 target="_blank">Orders',
 obj.id
)
```

And now if we reload the admin interface, we could see our new alert box: “Click below”... and our light button “Orders”.

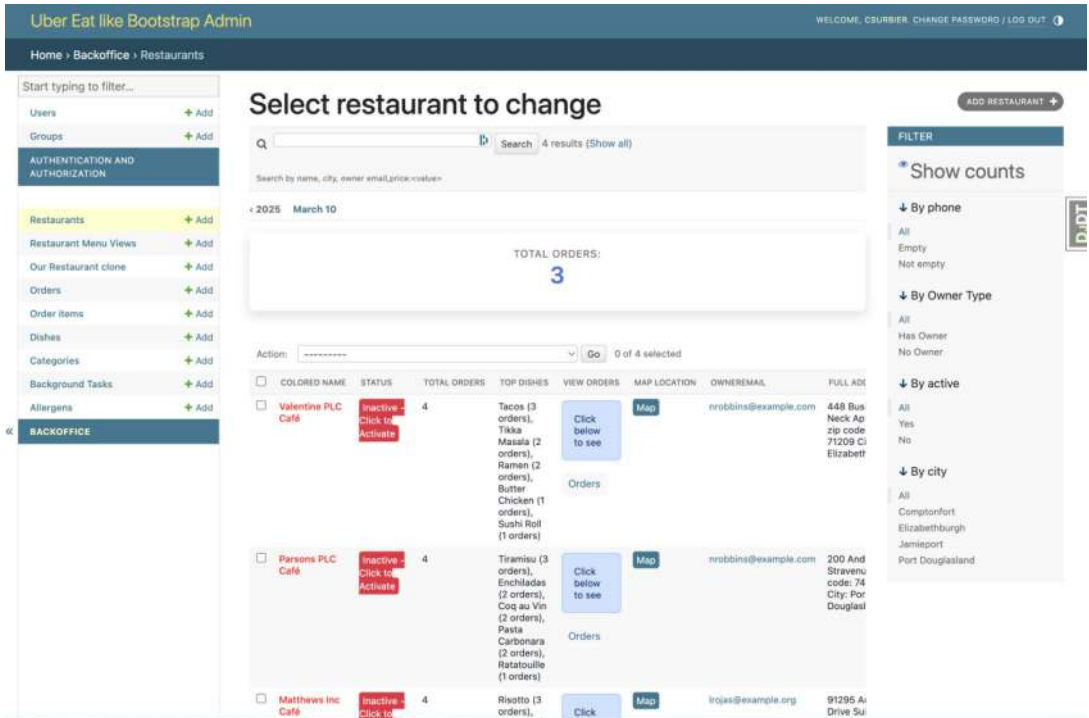


**Figure 16-1.** Bootstrap-styled alert and button in a custom `list_display` link column

## Bootstrap Locally

Or you can install **Bootstrap** locally, which is better for production (you are not dependent on a CDN, and files are directly installed on your server):

```
pip install django-bootstrap5==25.2
```

Then we need to edit our **settings.py** file to add the **django\_bootstrap5** app to our **INSTALLED\_APPS**:

```
INSTALLED_APPS = [
 "django_bootstrap5",
 'backoffice',
 'django.contrib.admin',
```

```

'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'debug_toolbar',
'silk',
'django_admin_inline_paginator',
]

```

And now we will load it by default. Let's modify our **base\_site.html** file:

```

<!-- templates/admin/base_site.html -->
{% extends "admin/base.html" %}
{% load static %}
{% load django_bootstrap5 %} <!-- This loads Bootstrap library-->

{% block title %}{{ title }} | {{ site_title|default:_('Django site admin') }}{% endblock %}

{% block extrahead %}
 {% bootstrap_css %} <!-- This loads Bootstrap CSS -->
{% endblock %}

{% block extrajs %}
 {% bootstrap_javascript %} <!-- This loads Bootstrap JS -->
{% endblock %}

{% block branding %}
<b1 id="site-name">
 Uber Eat like
 Bootstrap Admin
</b1>
{% endblock %}

```

If we reload our server and restaurant page, we should still have our Bootstrap elements.

# Tailwind CSS

To use Tailwind with our **Django Admin**, we will install **Django-Tailwind** package which is more easier:

```
pip install django-tailwind
pip install django_browser_reload==1.18.0
```

Then we need to add it to our `INSTALLED_APPS`:

```
INSTALLED_APPS = [
 'tailwind',
 'backoffice',
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'debug_toolbar',
 'silk',
 'django_admin_inline_paginator',
]
```

Then we need to init it, and once asked, we will create a theme named : **theme**:

```
python manage.py tailwind init
[1/2] Your Tailwind app/theme name (theme): theme
[2/2] Include daisyUI plugin?
1 - no
2 - yes
Choose from [1/2] (1):
Tailwind application 'theme' has been successfully created. Please add
'theme' to INSTALLED_APPS in settings.py, then run the following command to
install Tailwind CSS dependencies: `python manage.py tailwind install`
```

Then we need to add the **theme** to the list of our `INSTALLED_APPS` and also set the **TAILWIND\_APP\_NAME** and some options for **DEBUG** mode and also for static files configuration:

```
INSTALLED_APPS = [
 'tailwind',
 'theme',
 'backoffice',
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 'debug_toolbar',
 'silk',
 'django_admin_inline_paginator',
]

TAILWIND_APP_NAME = 'theme'

if DEBUG:
 # Add django_browser_reload only in DEBUG mode
 INSTALLED_APPS += ['django_browser_reload']
 MIDDLEWARE += ['django_browser_reload.middleware.
 BrowserReloadMiddleware']

Static file configuration
STATIC_URL = 'static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
STATICFILES_DIRS = [
 BASE_DIR / 'static',
]

Static file finders
STATICFILES_FINDERS = [
 'django.contrib.staticfiles.finders.FileSystemFinder',
 'django.contrib.staticfiles.finders.AppDirectoriesFinder',
]
```

To have the **django\_browser\_reload** work properly, we need to edit our **urls.py** file:

```
from django.conf import settings
from django.contrib import admin
from django.urls import include, path
from backoffice import views
from django.conf.urls.static import static
urlpatterns = [
 path('admin/', admin.site.urls),
 path('api/categories/', views.get_categories, name='get_categories'),
 path('api/get-category-dishes/', views.get_category_dishes, name='get-
category-dishes'),
 path('silk/', include('silk.urls', namespace='silk')),
]

if settings.DEBUG:
 urlpatterns += [path('__debug__/', include('debug_toolbar.urls'))]

if settings.DEBUG:
 # Include django_browser_reload URLs only in DEBUG mode
 urlpatterns += [
 path("__reload__/", include("django_browser_reload.urls")),
]
```

We will now modify our **base\_site.html** file to load tailwind:

```
{% extends "admin/base.html" %}
{% load static tailwind_tags %}
{% block title %}{% if subtitle %}{{ subtitle }} | {% endif %}{{ title }} |
{{ site_title|default:_('Django site admin') }}{% endblock %}

{% block extrahead %}
 {% tailwind_css %}
{% endblock %}

{% block branding %}
<div id="site-name">{{ site_
header|default:_('Django administration') }}</div>
{% if user.is_anonymous %}
```

```
{% include "admin/color_theme_toggle.html" %}
{% endif %}
{% endblock %}

{% block nav-global %}{% endblock %}
```

Now, to run it in development mode, we will not use our usual command **python manage.py runserver** but the specific command:

```
python manage.py tailwind install
```

Now let's modify our **view\_order** method of our **RestaurantAdmin** to check tailwind is running correctly:

```
@admin.display(description="View Orders")
def view_orders(self, obj):
 # Render a tailwind card with a button
 return format_html(
 '<div class="bg-white rounded-lg shadow p-6 mb-6">'
 '<h3 class="text-lg font-semibold mb-2">Orders</h3>'
 '<a class="bg-white-500 hover:bg-white-700 text-white font-bold'
 'py-2 px-4 rounded" href="/admin/backoffice/order/?restaurant__'
 'id_exact={}" target="_blank">View Orders'
 '</div>',
 obj.id
)
```

And voila, see Figure 16-2.



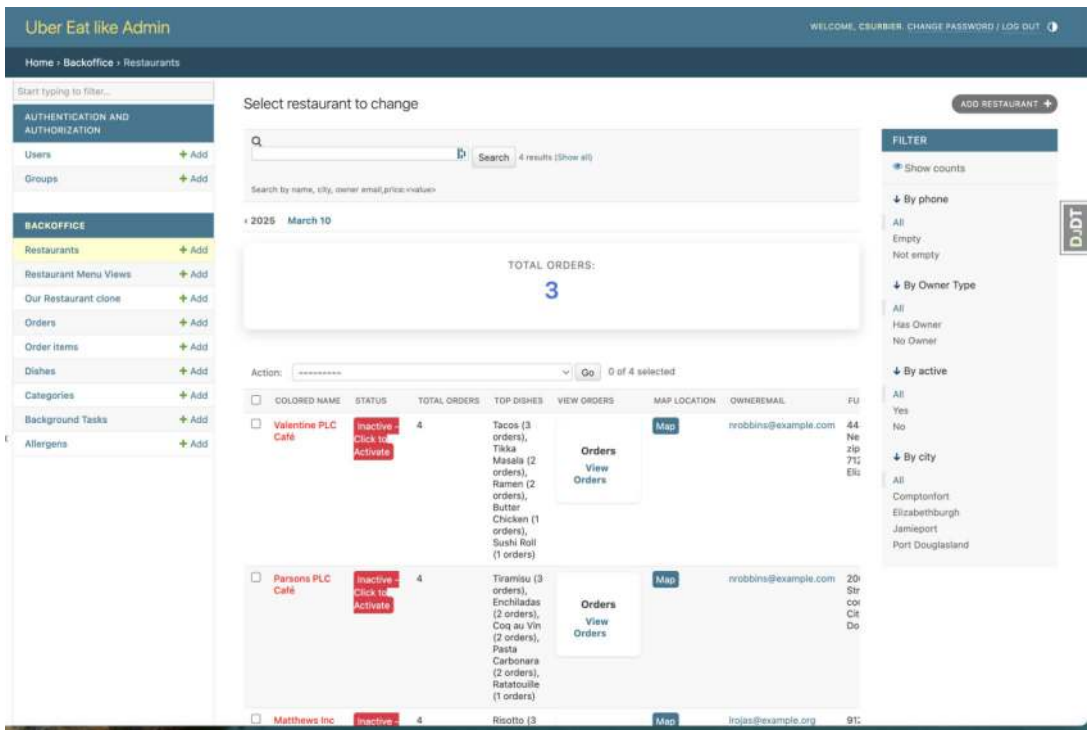


**Figure 16-2.** Tailwind CSS card and button in a custom `list_display` column

Please refer to [documentation \(https://django-tailwind.readthedocs.io/en/latest/index.html\)](https://django-tailwind.readthedocs.io/en/latest/index.html) to learn more and how to build for production.

## Material Design

If you want to include **Material design** in your Django Admin, you can use [Django material admin \(https://github.com/MaistrenkoAnton/django-material-admin\)](https://github.com/MaistrenkoAnton/django-material-admin).

---

You should now have a clear understanding of how to use a package extension with Django Admin, so we will not be covering this case.

---

## Custom Themes

Creating completely custom themes gives you full control over the admin's appearance, but it's quite a long, tedious, and sometimes even crazy job! My recommendation would be to use an existing Django Admin package, as we've seen before.... That said, let's see how to do it anyway!

In our application **backoffice**, we override some **Django Admin** templates:

```
backoffice/
├── templates/
│ └── admin/
│ ├── base_site.html
│ ├── change_list.html
│ └── index.html
│ ...
```

We will use the same behavior to override with custom CSS files. Like default templates, you will find **Django Admin** CSS and JavaScript files in the folder:

```
ls ../venv/lib/python3.12/site-packages/django/contrib/admin/static/
admin/css
```

To override or use our custom CSS, we will first create a new folder structure in our app **backoffice** which will contain

```
backoffice/
├── static/
│ ├── admin/
│ │ ├── css/
│ │ │ ├── mytheme.css
│ │ │ └── components/
│ │ │ └── myforms.css
│ │ └── js/
│ │ └── mytheme.js
├── templates/
│ └── admin/
│ ├── base_site.html
│ ├── index.html
│ └── components/
```

Let's create these directories and files:

```
mkdir -p backoffice/static/admin/css/components
mkdir backoffice/static/admin/js
touch backoffice/static/admin/css/mytheme.css
touch backoffice/static/admin/css/components/myforms.css
touch backoffice/static/admin/js/mytheme.js
```

Now let's start with our new theme file **mytheme.css**:

```
/* static/admin/css/theme.css */
:root {
 /* Color Palette */
 --primary-color: #9925be;
 --secondary-color: #598bad;
 --accent-color: #e74c3c;
 --success-color: #27ae60;
 --warning-color: #f39c12;
 --danger-color: #e74c3c;

 /* Neutral Colors */
 --background-color: #ecf0f1;
 --surface-color: #ffffff;
 --text-primary: #2c3e50;
 --text-secondary: #7f8c8d;
 --border-color: #bdc3c7;

 /* Spacing */
 --spacing-xs: 0.25rem;
 --spacing-sm: 0.5rem;
 --spacing-md: 1rem;
 --spacing-lg: 1.5rem;
 --spacing-xl: 2rem;

 /* Typography */
 --font-family-primary: 'Inter', -apple-system, BlinkMacSystemFont,
 sans-serif;
 --font-size-sm: 0.875rem;
```

```

--font-size-base: 1rem;
--font-size-lg: 1.125rem;
--font-size-xl: 1.25rem;

/* Shadows */
--shadow-sm: 0 1px 3px rgba(0,0,0,0.1);
--shadow-md: 0 4px 6px rgba(0,0,0,0.1);
--shadow-lg: 0 10px 15px rgba(0,0,0,0.1);

/* Border Radius */
--radius-sm: 0.25rem;
--radius-md: 0.375rem;
--radius-lg: 0.5rem;
}

/* Base Styles */
body.admin {
 font-family: var(--font-family-primary);
 background-color: var(--background-color);
 color: var(--text-primary);
 line-height: 1.6;
}

/* Header Styling */
#header {
 background: linear-gradient(135deg, var(--primary-color),
 var(--secondary-color));
 box-shadow: var(--shadow-md);
 border: none;
}

#branding h1 {
 color: white;
 font-weight: 600;
 font-size: var(--font-size-xl);
}

```

```

/* Navigation */
.nav-sidebar {
 background: var(--surface-color);
 box-shadow: var(--shadow-sm);
 border-radius: var(--radius-lg);
 margin: var(--spacing-md);
}

.nav-sidebar .current-app {
 background: var(--secondary-color);
 color: white;
 border-radius: var(--radius-md);
}

/* Content Area */
.content {
 padding: var(--spacing-xl);
}

.module {
 background: var(--surface-color);
 border-radius: var(--radius-lg);
 box-shadow: var(--shadow-sm);
 margin-bottom: var(--spacing-lg);
 overflow: hidden;
 border: 1px solid var(--border-color);
}

.module h2 {
 background: var(--primary-color);
 color: white;
 padding: var(--spacing-md) var(--spacing-lg);
 font-weight: 500;
 font-size: var(--font-size-lg);
}

```

In that file, we are overriding some **Django Admin** default values (which could be found in the default `base.css` file). Now the same idea, we will override some default forms values. Let's create our *components/myforms.css* new file:

```
/* static/admin/css/components/forms.css */
.form-container {
 background: var(--surface-color);
 border-radius: var(--radius-lg);
 padding: var(--spacing-xl);
 box-shadow: var(--shadow-sm);
}

.fieldset {
 border: 1px solid var(--border-color);
 border-radius: var(--radius-md);
 padding: var(--spacing-lg);
 margin-bottom: var(--spacing-lg);
}

.fieldset h2 {
 background: var(--background-color);
 color: var(--text-primary);
 padding: var(--spacing-sm) var(--spacing-md);
 margin: calc(-1 * var(--spacing-lg)) calc(-1 * var(--spacing-lg))
 var(--spacing-md);
 border-bottom: 1px solid var(--border-color);
 font-size: var(--font-size-base);
 font-weight: 600;
}

.form-row {
 margin-bottom: var(--spacing-md);
}

.form-row label {
 display: block;
 font-weight: 500;
 color: var(--text-primary);
}
```

```

 margin-bottom: var(--spacing-xs);
}

.form-row input,
.form-row select,
.form-row textarea {
 width: 100%;
 padding: var(--spacing-sm) var(--spacing-md);
 border: 1px solid var(--border-color);
 border-radius: var(--radius-md);
 font-size: var(--font-size-base);
 transition: all 0.2s ease;
}

.form-row input:focus,
.form-row select:focus,
.form-row textarea:focus {
 outline: none;
 border-color: var(--secondary-color);
 box-shadow: 0 0 3px rgba(52, 152, 219, 0.1);
}

```

and our **mytheme.js** file:

```

// static/admin/js/theme.js
document.addEventListener('DOMContentLoaded', function() {
 console.log("Custom Django Admin JS loaded!");

 // Example: change background color of save button
 document.querySelectorAll('input[type=submit].default').
 forEach(btn => {
 btn.style.backgroundColor = '#9925be';
 });

 // Add smooth transitions
 const elements = document.querySelectorAll('.module, .button, .form-row
 input');
 elements.forEach(el => {

```

```

 el.style.transition = 'all 0.2s ease';
 });

 // Enhanced form validation styling
 const forms = document.querySelectorAll('form');
 forms.forEach(form => {
 const inputs = form.querySelectorAll('input, select, textarea');
 inputs.forEach(input => {
 input.addEventListener('blur', function() {
 if (this.value.trim() !== '') {
 this.classList.add('has-value');
 } else {
 this.classList.remove('has-value');
 }
 });
 });
 });
});

```

Now we will modify our **base\_site.html** file to use these files:

```

{% extends "admin/base.html" %}
{% load static %}

{% block title %}{% if subtitle %}{{ subtitle }} | {% endif %}{{ title }} |
{{ site_title|default:_('Django site admin') }}{% endblock %}
{% block extrahead %}
 {{ block.super }}
 <link rel="stylesheet" href="{% static 'admin/css/mytheme.css' %}">
 <script src="{% static 'admin/js/mytheme.js' %}?v=1"></script>
{% endblock %}

{% block branding %}
<div id="site-name">{{ site_
header|default:_('Django administration') }}</div>
{% if user.is_anonymous %}
 {% include "admin/color_theme_toggle.html" %}

```



```
{% endif %}
{% endblock %}
{% block nav-global %}
{% endblock %}
```

And now if we reload the server, we should see our custom theme as shown from Figure 16-3 to Figure 16-5.

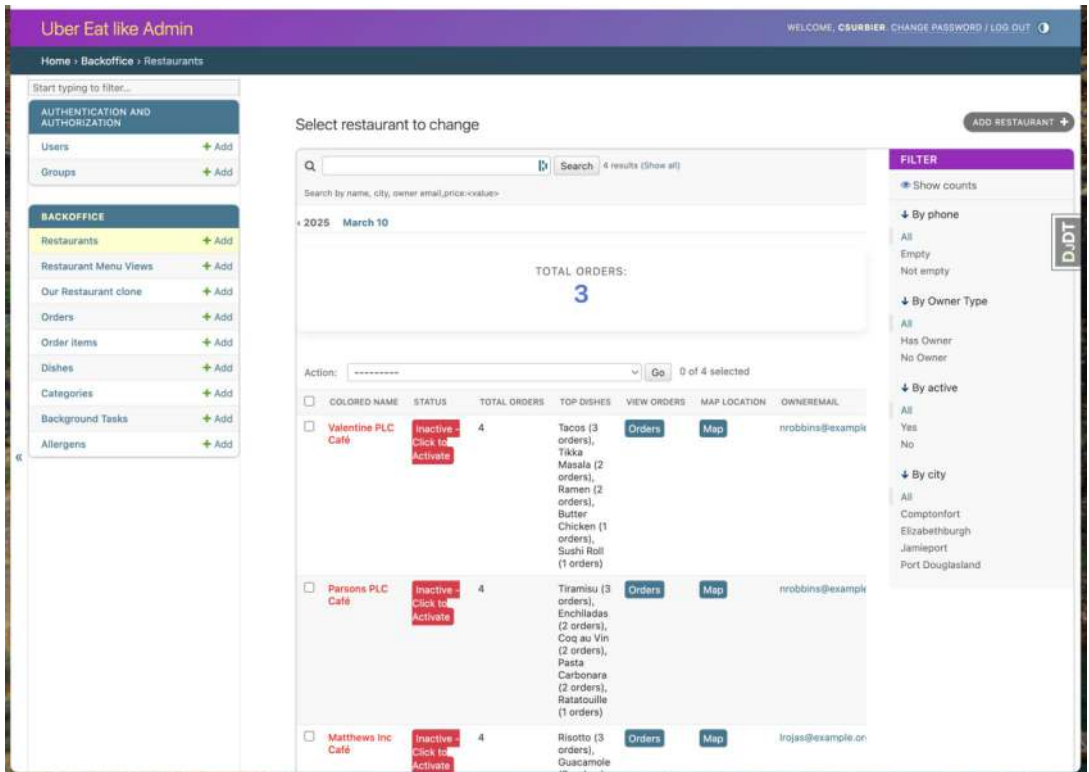


**Figure 16-3.** Custom admin theme: main interface with updated colors and layout

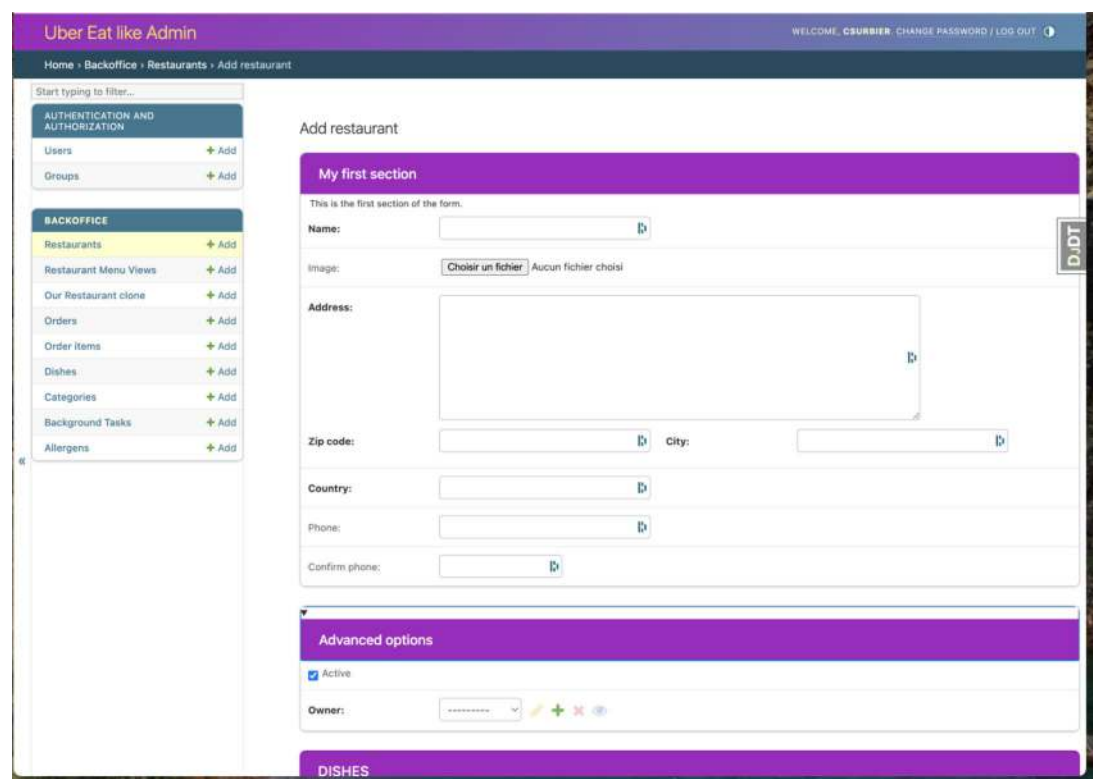


**Figure 16-4.** Custom theme applied to admin change forms

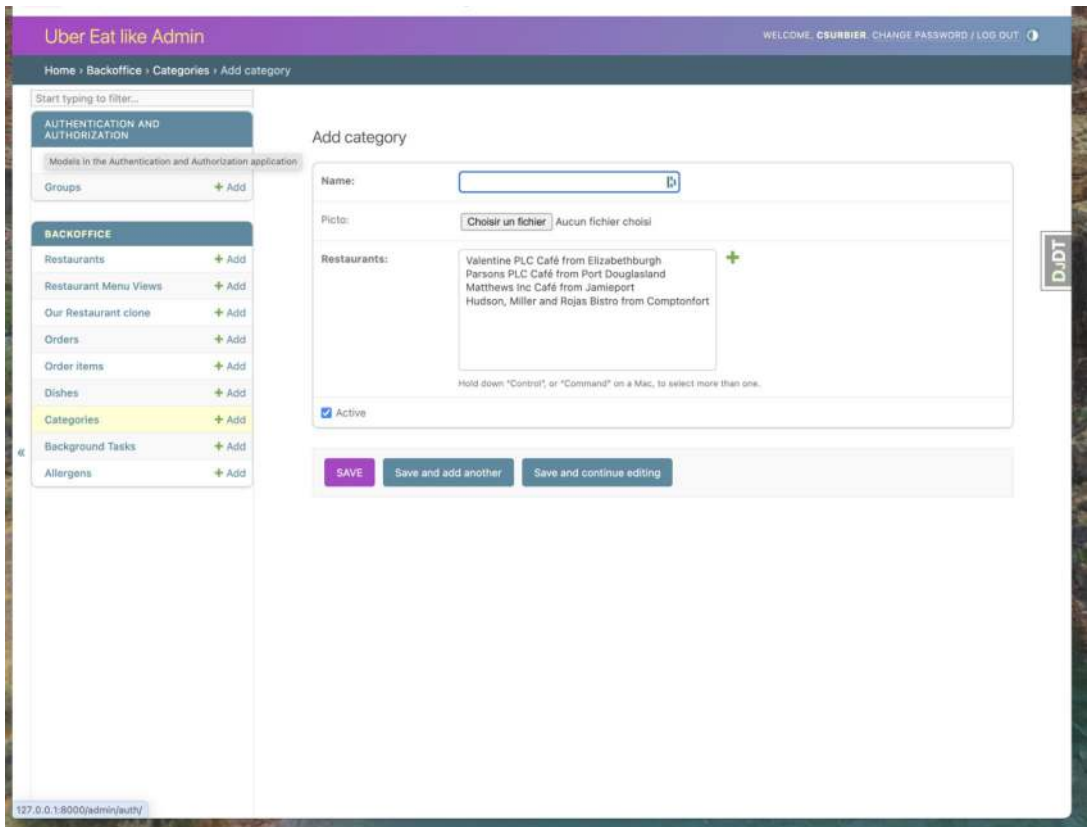


**Figure 16-5.** Custom JavaScript-driven behavior in the themed admin

Nice. Now you have got the idea. To change the color of the sidebar at the left (see Figure 16-6), we could modify our **mytheme.css** file and add

```
/* Full-width header styling */
#nav-sidebar caption {
 background-color: #9925be !important;
}

#nav-sidebar caption a {
 color: white !important;
}
```

```

/* Sidebar app section headers (e.g., BACKOFFICE, AUTHENTICATION...) */
#nav-sidebar .app-list .section {
 background-color: #9925be !important;
 color: white !important;
}

/* Sidebar links */
#nav-sidebar .model a:link,
#nav-sidebar .model a:visited {
 color: #9925be !important; /* link text color */
 font-weight: 500;
}

/* Sidebar link hover */
#nav-sidebar .model a:hover {
 color: white !important; /* text turns white */
 background-color: #9925be !important; /* highlight background */
}

/* "Add" green plus icon override */
#nav-sidebar .addlink {
 color: #00ff99 !important; /* pick your accent color here */
}

/* "Change" icon override (optional) */
#nav-sidebar .changelink {
 color: #999 !important;
}

```

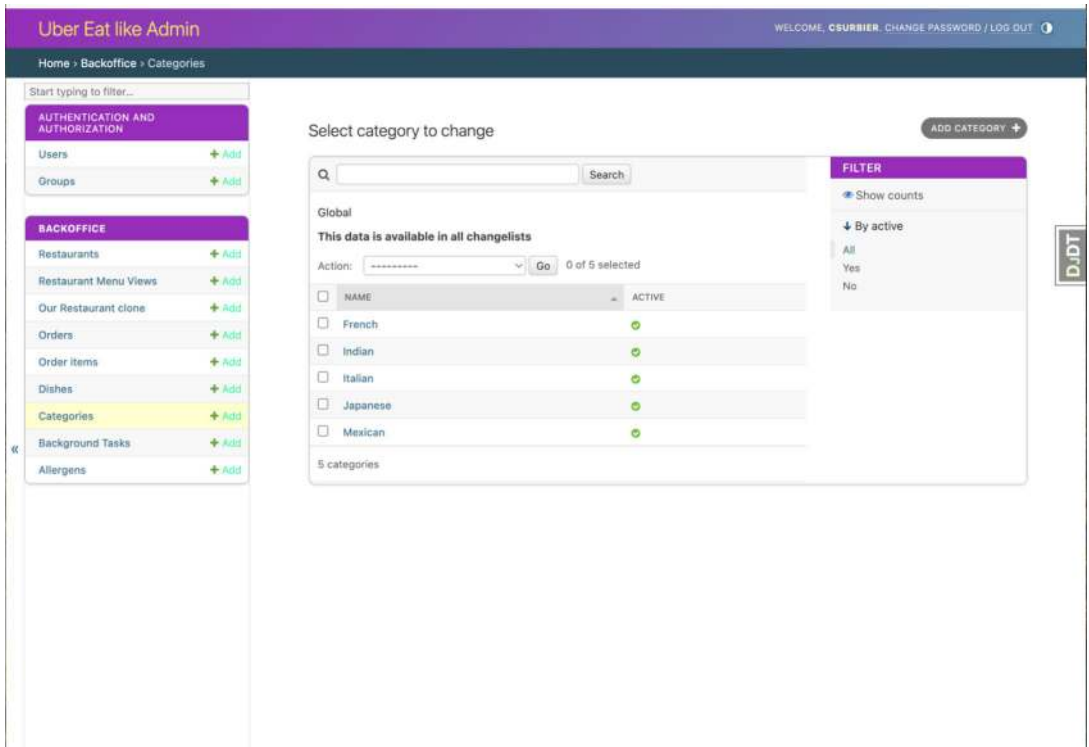


**Figure 16-6.** Custom sidebar styling (nav sidebar colors and links)

And to find the variable names to override, we need to look in the `../venv/lib/python3.12/site-packages/django/contrib/admin/static/admin/css/nav_sidebar.css` default Django Admin file (or completely replace this file).

That's it; now we know the basics to fully customize our **Django Admin** interface, but as I said, this is a tedious job!

## CHAPTER 17

# JavaScript Integration

---

In the next chapters, we will use our custom theme but with **Bootstrap** added to our **base\_site.html** file as we learned previously:

---

```
{% extends "admin/base.html" %}
{% load static %}
{% load django_bootstrap5 %} <!-- This loads Bootstrap library-->
{% block title %}{% if subtitle %}{{ subtitle }} | {% endif %}{{ title }} |
{{ site_title|default:_('Django site admin') }}{% endblock %}
{% block extrahead %}
 {{ block.super }}
 <link rel="stylesheet" href="{% static 'admin/css/mytheme.css' %}">
 {% bootstrap_css %} <!-- This loads Bootstrap CSS -->
 <script src="{% static 'admin/js/mytheme.js' %}?v=1"></script>
{% endblock %}
{% block extrajs %}
 {% bootstrap_javascript %} <!-- This loads Bootstrap JS -->
{% endblock %}

{% block branding %}
<div id="site-name">{{ site_
header|default:_('Django administration') }}</div>
{% if user.is_anonymous %}
 {% include "admin/color_theme_toggle.html" %}
{% endif %}
{% endblock %}
{% block nav-global %}
{% endblock %}
```

## HTMX and Ajax Operations

In Chapter 3, we have already seen how to integrate Ajax operations in our admin. Let's go further and see how to integrate HTMX in our **Django Admin**. As a reminder:

---

HTMX is a lightweight JavaScript library that lets you make parts of a web page update dynamically—without reloading the whole page—just by adding special HTML attributes. It can handle things like fetching new content, submitting forms, or swapping HTML fragments via Ajax, WebSockets, or Server-Sent Events, all without writing any custom JavaScript. Think of it as: “Ajax without the JavaScript boilerplate—just HTML attributes.”

---

Let's modify our **base\_site.html** to load HTMX library:

```
...
{% block extrahead %}
 {{ block.super }}
 <link rel="stylesheet" href="{% static 'admin/css/mytheme.css' %}">
 {% bootstrap_css %} <!-- This loads Bootstrap CSS -->
 <script src="{% static 'admin/js/mytheme.js' %}?v=1"></script>
 <script src="https://unpkg.com/htmx.org@1.9.10"></script> <!-- This
 loads HTMX library -->
{% endblock %}
...
```

Our main **Django Admin** page will look as shown in Figure 17-1.

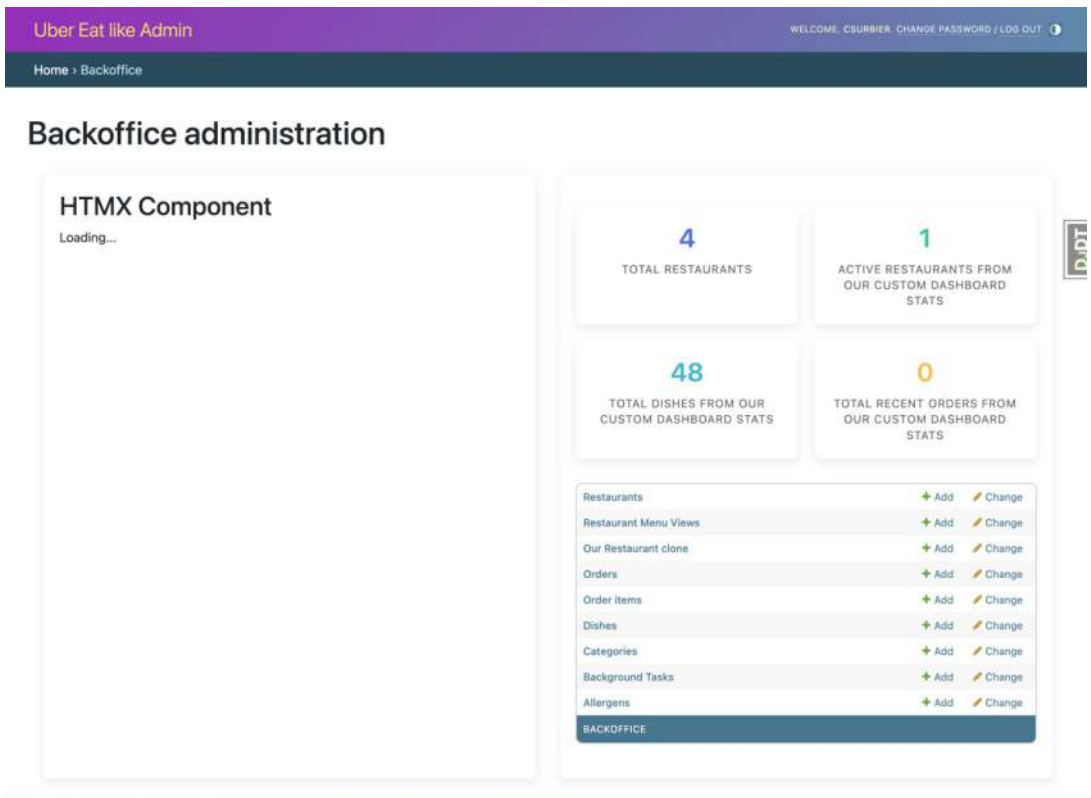


**Figure 17-1.** Admin index with HTMX column alongside dashboard stats and app list

We will have our HTMX component on the left side and our usual statistics and backoffice items menu on the right side. Let's modify our custom **index.html** file to create this design:

```
{% extends "admin/base_site.html" %}
{% load i18n static admin_extras %}

{% block extrastyle %}{{ block.super }}<link rel="stylesheet" href="{%
static "admin/css/dashboard.css" %}">
<style>
 .dashboard-stats {
 display: flex;
 flex-wrap: wrap;
 gap: 20px;
```



```

 margin: 20px 0 30px;
 }

.stat-card {
 flex: 1;
 min-width: 200px;
 background: white;
 border-radius: 8px;
 box-shadow: 0 4px 12px rgba(0, 0, 0, 0.08);
 padding: 20px;
 text-align: center;
 transition: transform 0.3s ease, box-shadow 0.3s ease;
}

.stat-card:hover {
 transform: translateY(-5px);
 box-shadow: 0 6px 16px rgba(0, 0, 0, 0.12);
}

.stat-card h3 {
 font-size: 32px;
 margin: 0 0 10px;
 color: #417690;
 font-weight: 600;
}

.stat-card p {
 margin: 0;
 color: #666;
 font-size: 14px;
 text-transform: uppercase;
 letter-spacing: 1px;
}

/* Color variations for different cards */
.stat-card:nth-child(1) h3 { color: #4e73df; }
.stat-card:nth-child(2) h3 { color: #1cc88a; }
.stat-card:nth-child(3) h3 { color: #36b9cc; }

```

```

.stat-card:nth-child(4) h3 { color: #f6c23e; }

/* Override Django admin's layout constraints */
body.dashboard #container {
 width: 100vw !important;
 max-width: none !important;
 margin: 0 !important;
}

body.dashboard #content {
 width: 100% !important;
 max-width: none !important;
 margin: 0 !important;
 padding: 0 !important;
}

body.dashboard #content-main {
 width: 100% !important;
 max-width: none !important;
 margin: 0 !important;
 padding: 20px !important;
}

/* Two-column layout using flexbox */
.dashboard-container {
 display: flex;
 gap: 30px;
 width: 100%;
 margin: 0;
 min-height: 400px;
}

.htmx-column,
.dashboard-column {
 flex: 1;
 background: white;
 border-radius: 8px;
 box-shadow: 0 4px 12px rgba(0, 0, 0, 0.08);
}

```

```

 padding: 20px;
}

/* Responsive adjustments */
@media (max-width: 768px) {
 .dashboard-container {
 grid-template-columns: 1fr;
 gap: 20px;
 }

 .dashboard-stats {
 flex-direction: column;
 }

 .stat-card {
 margin-bottom: 15px;
 }
}
</style>

{% endblock %}

{% block coltype %}colMS{% endblock %}

{% block bodyclass %}{{ block.super }} dashboard{% endblock %}

{% block nav-breadcrumbs %}{% endblock %}

{% block nav-sidebar %}{% endblock %}

{% block content %}
<div id="content-main">
 <div class="dashboard-container">
 <!-- First column: HTMX component -->
 <div class="htmx-column">
 <h2>HTMX Component</h2>
 <!-- Add your HTMX component here -->
 <div hx-get="/our-htmx-endpoint" hx-trigger="load">
 Loading...

```

```

 </div>
</div>

<!-- Second column: Existing dashboard -->
<div class="dashboard-column">
 <div class="dashboard-stats">
 <div class="stat-card">
 <h3>{% total_restaurants %}</h3>
 <p>Total Restaurants</p>
 </div>
 <!-- Using inclusion tag for complex widgets -->
 {% dashboard_statistics %}
 </div>
 {% include "admin/app_list.html" with app_list=app_list show_
 changelinks=True %}
</div>
</div>
</div>
{% endblock %}

{% block sidebar %}
<div id="content-related">
 <div class="module" id="recent-actions-module">
 <h2>{% translate 'Recent actions' %}</h2>
 <h3>{% translate 'My actions' %}</h3>
 {% load log %}
 {% get_admin_log 10 as admin_log for_user user %}
 {% if not admin_log %}
 <p>{% translate 'None available' %}</p>
 {% else %}
 <ul class="actionlist">
 {% for entry in admin_log %}
 <li class="{% if entry.is_addition %}addlink{% endif %}{% if
 entry.is_change %}changelink{% endif %}{% if entry.is_deletion
 %}deletelink{% endif %}">

```

```

 {% if entry.is_addition %}{%
 translate 'Added:' %}{% elif entry.is_change %}{% translate
 'Changed:' %}{% elif entry.is_deletion %}{% translate
 'Deleted:' %}{% endif %}
 {% if entry.is_deletion or not entry.get_admin_url %}
 {{ entry.object_repr }}
 {% else %}
 {{ entry.object_
 repr }}
 {% endif %}

 {% if entry.content_type %}
 {% filter capfirst %}{{ entry.
 content_type.name }}{% endfilter %}
 {% else %}
 {% translate 'Unknown content'
 %}
 {% endif %}

 {% endfor %}

{% endif %}
</div>
</div>
{% endblock %}

```

Our goal will be to replace the HTMX column:

```

<div hx-get="/our-hmx-endpoint" hx-trigger="load">
 Loading...
</div>

```

With the list of recent orders, it will be very useful to see in real time the new orders a restaurant receives. To achieve this goal, we will need to

- Create a view for recent orders HTMX endpoint
- Add a URL pattern for recent orders endpoint

- Create a template for recent orders display
- Update HTMX component that will use our new endpoint with an auto-refresh instruction

## Create a View with Recent Orders for Our HTMX Component

Let's add a new method in our **views.py** file:

```
@staff_member_required
def recent_orders(request):
 """
 HTMX view to get recent orders with auto-refresh
 """
 orders = Order.objects.select_related('customer', 'restaurant').order_
by('-created_at')[:5]

 return render(request, 'admin/recent_orders.html', {
 'orders': orders
 })
```

Really simple, the view will get the five last orders and will render the result with the **admin/recent\_orders.html** template that we will create soon.

## Add URLs

Now we need to register this new view in our main **urls.py** file:

```
from django.conf import settings
from django.contrib import admin
from django.urls import include, path
from backoffice import views
from django.conf.urls.static import static
urlpatterns = [
 path('admin/', admin.site.urls),
 path('api/categories/', views.get_categories, name='get_categories'),
```

```

 path('api/get-category-dishes/', views.get_category_dishes, name='get-
category-dishes'),
 path('api/recent-orders/', views.recent_orders, name='recent-orders'),
 path('silk/', include('silk.urls', namespace='silk')),
]

```

**if** settings.DEBUG:

```
 urlpatterns += [path('__debug__/', include('debug_toolbar.urls'))]
```

So our endpoint will be /api/recent-orders/.

## Create a Template for Recent Orders Display

Now let's create our template file to render our orders:

```
touch backoffice/templates/admin/recent_orders.html
```

and then write the code:

```

<div class="recent-orders">
 <h3>Recent Orders</h3>
 {% if orders %}
 <div class="orders-list">
 {% for order in orders %}
 <div class="order-item">
 <div class="order-header">
 Order #{{ order.idslice:"8" }}...
 <span class="order-status status-{{ order.status
 }}">{{ order.get_status_display }}
 </div>
 <div class="order-details">
 <p>Customer: {{ order.customer.
 username }}</p>
 <p>Restaurant: {{ order.
 restaurant.name }}</p>
 <p>Total: ${{ order.total_
 amount }}</p>
 </div>
 </div>
 {% endfor %}
 </div>
 {% else %}
 <p>No recent orders found.</p>
 {% endif %}
</div>

```

```

 <p class="order-time">{{ order.created_at|timesince
 }} ago</p>
 </div>
</div>
{% endfor %}
</div>
{% else %}
 <p class="no-orders">No recent orders</p>
{% endif %}
</div>
<style>
.recent-orders {
 height: 100%;
 display: flex;
 flex-direction: column;
}

.recent-orders h3 {
 margin-top: 0;
 color: #417690;
 border-bottom: 2px solid #e9ecef;
 padding-bottom: 10px;
 flex-shrink: 0;
}

.orders-list {
 flex: 1;
 overflow-y: auto;
 min-height: 0;
}

.order-item {
 border: 1px solid #e9ecef;
 border-radius: 6px;
 margin-bottom: 10px;
 padding: 15px;

```



```

 background: #f8f9fa;
 transition: all 0.2s ease;
}

.order-item:hover {
 background: #e9ecef;
 transform: translateX(5px);
}

.order-header {
 display: flex;
 justify-content: space-between;
 align-items: center;
 margin-bottom: 10px;
}

.order-status {
 padding: 4px 8px;
 border-radius: 4px;
 font-size: 12px;
 font-weight: bold;
 text-transform: uppercase;
}

.status-pending { background: #fff3cd; color: #856404; }
.status-confirmed { background: #d4edda; color: #155724; }
.status-preparing { background: #cce5ff; color: #004085; }
.status-delivering { background: #e7f3ff; color: #0066cc; }
.status-delivered { background: #d1ecf1; color: #0c5460; }
.status-cancelled { background: #f8d7da; color: #721c24; }

.order-details p {
 margin: 5px 0;
 font-size: 13px;
}

```

```

.order-time {
 color: #6c757d;
 font-style: italic;
}

.no-orders {
 text-align: center;
 color: #6c757d;
 font-style: italic;
 padding: 40px 0;
}
</style>

```

## Update the HTMX Component with Our New Endpoint and with an Auto-Refresh Instruction

And now we have all the elements to update our **index.html** file and use our new endpoint that will render the latest orders:

```

<div class="htmx-column">
 <div hx-get="{% url 'recent-orders' %}"
 hx-trigger="load, every 10s"
 hx-indicator="#loading">
 <div id="loading" class="htmx-indicator">Loading recent
 orders...</div>
 </div>
</div>

```

This is quite magical, but just with that, HTMX framework will call our new API endpoint each ten seconds and will replace the **div** with id="loading" with the template results.

Figure 17-2 shows the final result.

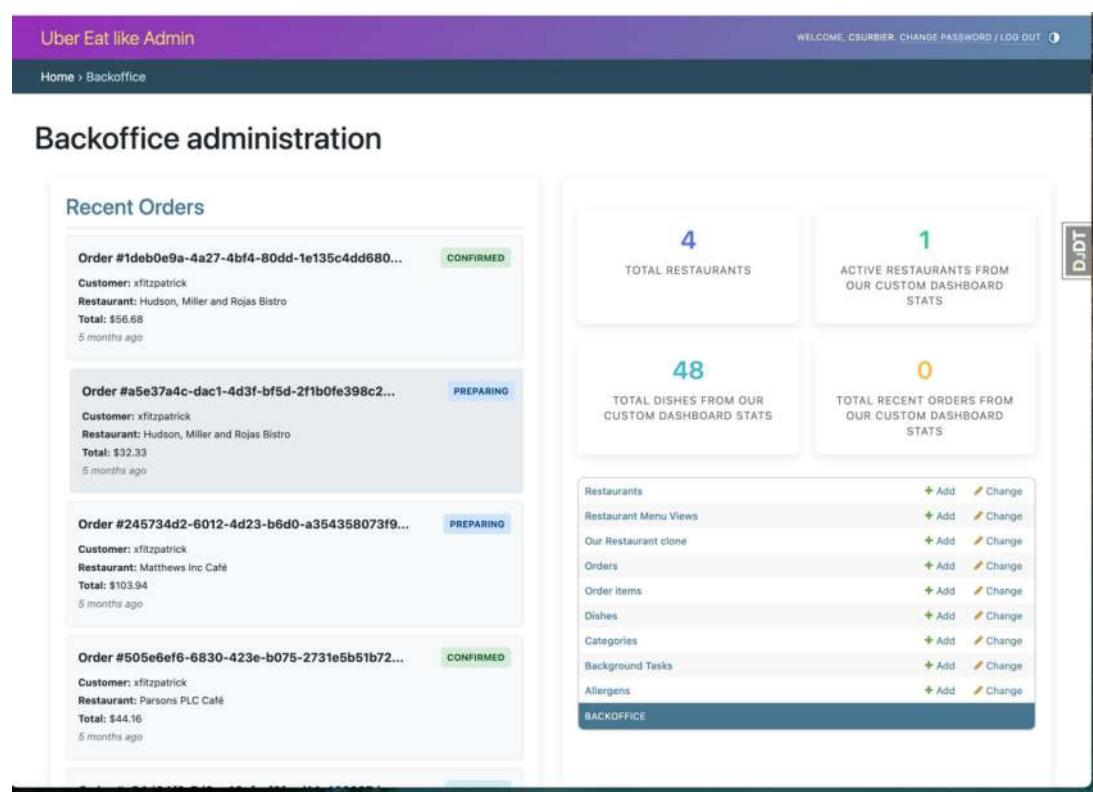


**Figure 17-2.** Recent orders panel on the admin index loaded via HTMX

We now have our most recent orders directly displayed on our main backoffice page, with an automatic refresh.

We can verify the auto-refresh is working correctly just by adding a new order. To do so, open a new tab browser (to keep the current page) and add a new order (Figure 17-3).

The screenshot shows the 'Add order' form in the 'Uber Eat like Admin' application. The sidebar on the left contains a search bar and a list of menu items: Users, Groups, AUTHENTICATION AND AUTHORIZATION, Restaurants, Restaurant Menu Views, Our Restaurant clone, Orders (highlighted), Order items, Dishes, Categories, Background Tasks, Allergens, and BACKOFFICE. The main form has the following fields:

- Customer:** A dropdown menu with 'csurbler' selected.
- Restaurant:** A dropdown menu with 'Valentine PLC Café from Elizabethburgh' selected.
- Status:** A dropdown menu with 'Confirmed' selected.
- Delivery address:** A text area with 'My adresse please' entered.
- Total amount:** A numeric input field with '20' entered.

At the bottom of the form, there are three buttons: 'SAVE', 'Save and add another', and 'Save and continue editing'. A 'D:JDT' label is visible on the right side of the form.

**Figure 17-3.** Adding a new order in a separate tab to test HTMX refresh

Save it and then go back to your main page tab; you should or will see the new order coming into the list (Figure 17-4).

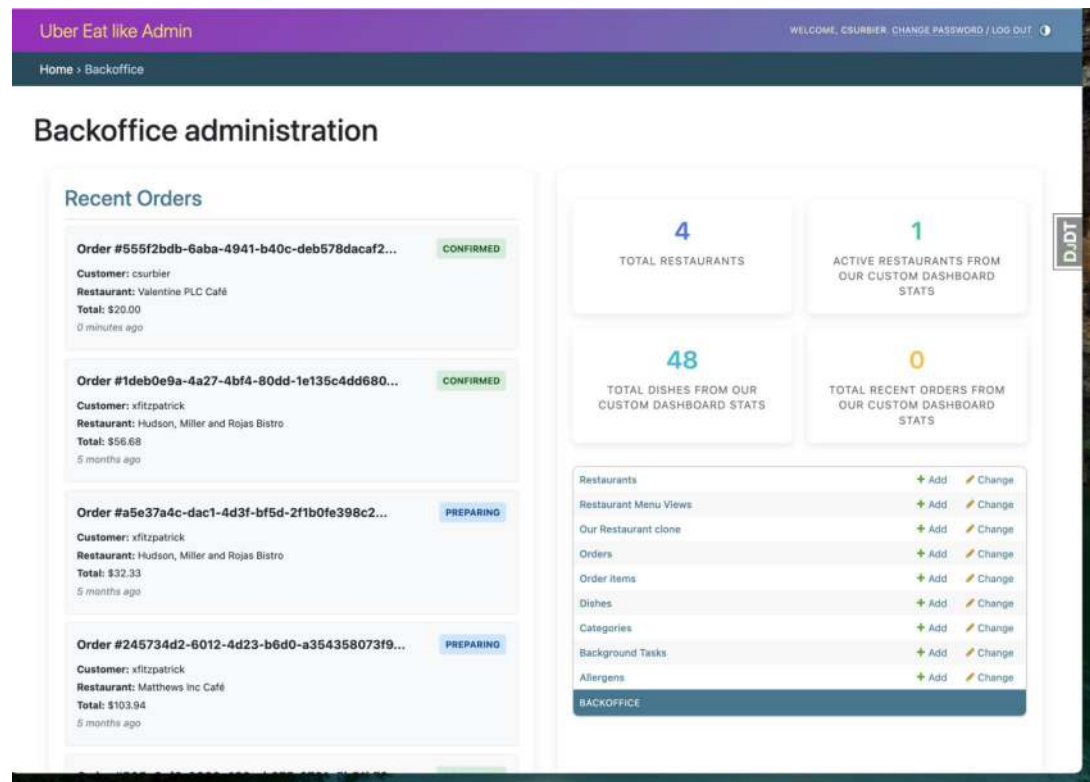


**Figure 17-4.** Recent orders list updated after automatic HTMX refresh

---

And indeed we have the new order from **csurbier** in the list. Amazing!

---

## CHAPTER 18

# Responsive Design

Modern Django Admin is already mobile-friendly, but when customizing it, you should ensure your changes preserve—or even improve—its responsiveness. This is especially important if your users need to manage data on tablets or smartphones.

## Mobile-First Approach

Design and test your admin customizations starting with small screens first. This ensures essential actions remain accessible and extra features are added progressively for larger viewports.

## Adaptive Layouts

Use flexible grids (CSS Grid or Flexbox) so elements automatically adjust to different screen sizes. Avoid fixed widths that can break layouts on narrow devices.

## Media Queries

Leverage CSS media queries to fine-tune styles for specific breakpoints. For example, hide secondary navigation on smaller screens or adjust table layouts for better readability.

As a good example, we can have a look at our new index page with its two columns. On mobile, it's not responsive, and we can see the beginning of the second column as shown on Figure 18-1.

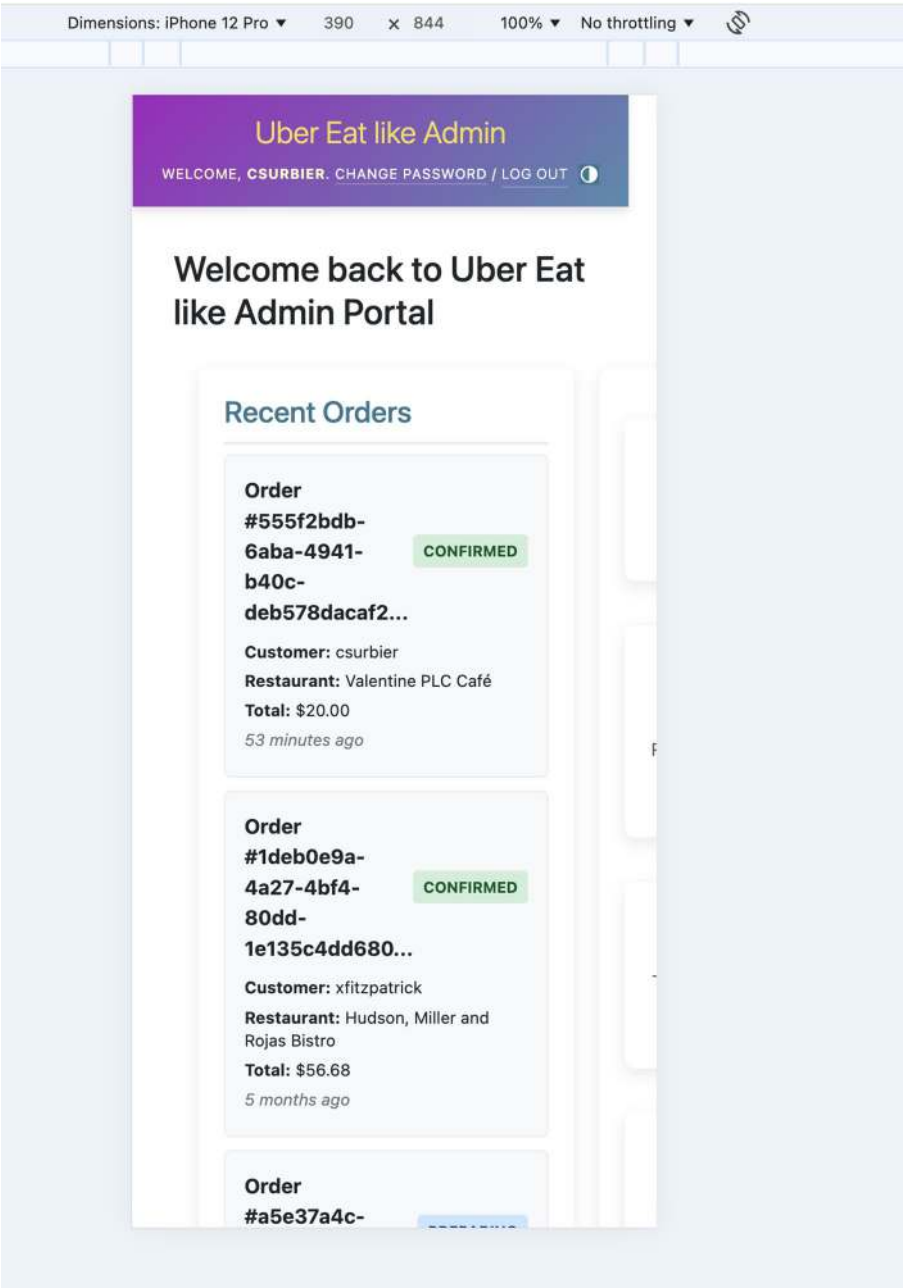


**Figure 18-1.** Two-column admin dashboard layout breaking on narrow viewports (before responsive fix)

We need to adapt our CSS with media queries to manage mobile screen (see Figure 18-2). Let's edit our **index.html** file and add

```
/* Responsive adjustments */
@media (max-width: 768px) {
 .dashboard-container {
 flex-direction: column;
 gap: 20px;
 }

 .htmz-column,
 .dashboard-column {
 min-height: 300px;
 }

 .dashboard-stats {
 flex-direction: column;
 }

 .stat-card {
 margin-bottom: 15px;
 }
}
```



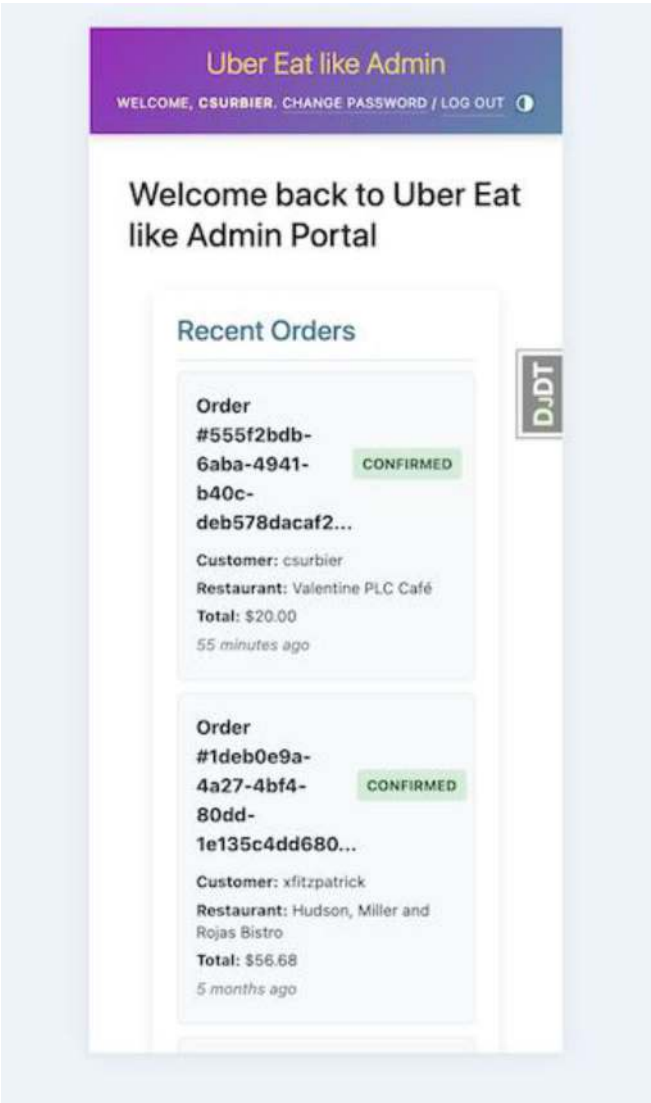


**Figure 18-2.** *Stacked single-column layout on mobile after media-query adjustments*

Now our index page will stack our columns vertically on mobile screen (i.e.,  $\leq 768\text{px}$ ).

## CHAPTER 19

# Custom Dashboards

By modifying the default **Django Admin** index page, we already have implemented a custom dashboard. This is quite common with **Django Admin** to build its own custom dashboard (see Figure 19-1), because

- It provides quick, actionable insights (sales totals, recent orders, pending reviews)
- It reduces clicks for common admin tasks
- It can integrate real-time updates via HTMX or WebSockets as we have done previously

As another example, we will create simple TODO widgets to manage tasks directly from our admin dashboard (see Figure 19-2). The widget will be composed of a form to add a new task and a list of tasks.

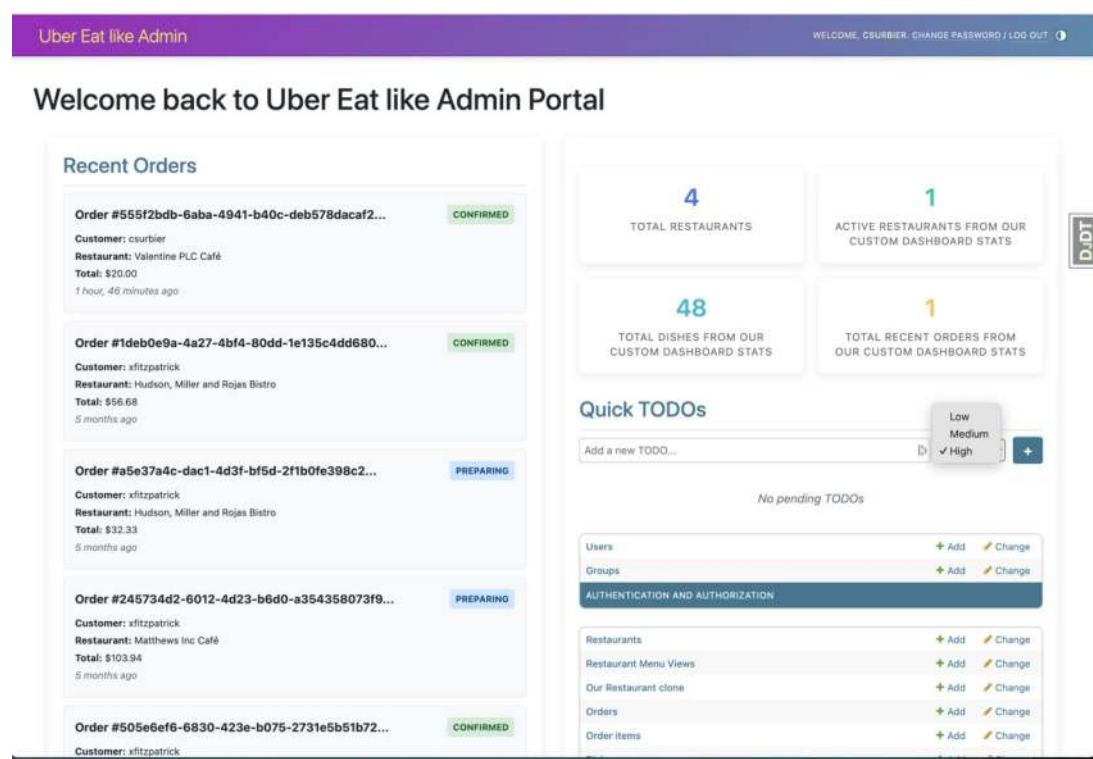


Figure 19-1. Custom dashboard TODO widget (form to add tasks)

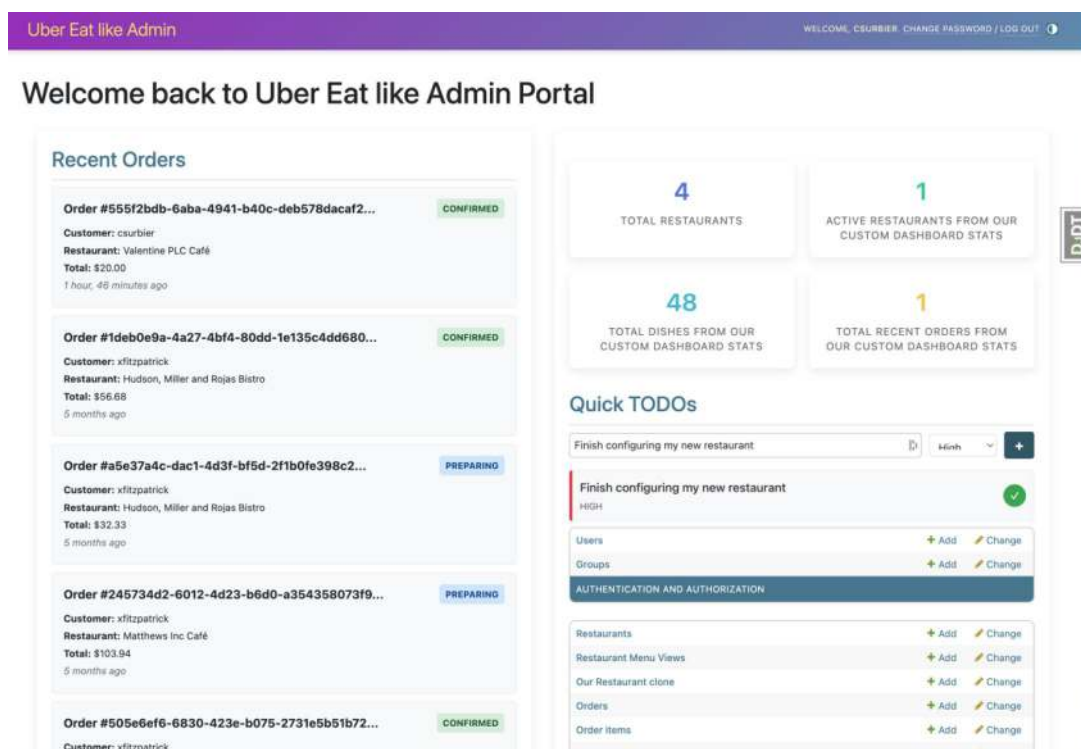


*Figure 19-2. TODO widget showing the updated task list*

## TODO Models

First, we will create a TODO model in `models.py`:

```
class Todo(models.Model):
 PRIORITY_CHOICES = [
 ('low', _('Low')),
 ('medium', _('Medium')),
 ('high', _('High')),
]

 id = models.UUIDField(primary_key=True, default=uuid.uuid4,
 editable=False)
 title = models.CharField(_('title'), max_length=200)
 description = models.TextField(_('description'), blank=True)
```

```

priority = models.CharField(_('priority'), max_length=10,
choices=PRIORITY_CHOICES, default='medium')
is_completed = models.BooleanField(_('completed'), default=False)
created_by = models.ForeignKey(User, on_delete=models.CASCADE, related_
name='todos')
created_at = models.DateTimeField(auto_now_add=True)
completed_at = models.DateTimeField(null=True, blank=True)

class Meta:
 verbose_name = _('todo')
 verbose_name_plural = _('todos')
 ordering = ['-created_at']

def __str__(self):
 return self.title

```

Our model will have a priority field to indicate the urgency of the task and an `is_completed` field to indicate if the task is completed or not.

And let's register TODO models in the **admin.py** file:

```

@admin.register(Todo)
class TodoAdmin(admin.ModelAdmin):
 list_display = ('title', 'description', 'is_completed')
 search_fields = ('title', 'description')
 list_filter = ('is_completed',)
 list_per_page = 10
 list_max_show_all = 200
 show_full_result_count = False
 date_hierarchy = 'created_at'

```

We need to run migrations to apply the changes to the database:

```

python manage.py makemigrations
python manage.py migrate

```

## Create Views for TODO Widget

Next, we need to create new endpoints and views to manage the widget. First, we will add a view to be able to add TODO:

```
@staff_member_required
@require_http_methods(["POST"])
def add_todo(request):
 """
 HTMX view to add a new TODO
 """
 title = request.POST.get('title', '').strip()
 priority = request.POST.get('priority', 'medium')

 if title:
 Todo.objects.create(
 title=title,
 priority=priority,
 created_by=request.user
)

 # Return updated TODO list
 todos = Todo.objects.filter(created_by=request.user,
 is_completed=False).order_by('-created_at')[:10]
 return render(request, 'admin/todo_list.html', {
 'todos': todos
 })
```

---

Please note the `@staff_member_required` decorator is used to restrict access to this view to staff members only.

---

Then, we will add a new view to return a list of TODO to our widgets:

```
@staff_member_required
def todo_widget(request):
 """
 HTMX view to display TODO widget
 """
 todos = Todo.objects.filter(created_by=request.user, is_
 completed=False).order_by('-created_at')[:10]
 return render(request, 'admin/todo_widget.html', {
 'todos': todos
 })
```

---

We are filtering the TODOs to only show the ones that are not completed and are created by the current user.

---

And finally, we will add a view to mark a TODO as completed:

```
@staff_member_required
@require_http_methods(["POST"])
def complete_todo(request, todo_id):
 """
 HTMX view to mark TODO as completed
 """
 todo = Todo.objects.get_object_or_404(id=todo_id, created_
 by=request.user)
 todo.is_completed = True
 todo.completed_at = timezone.now()
 todo.save()

 # Return updated TODO list
 todos = Todo.objects.filter(created_by=request.user, is_
 completed=False).order_by('-created_at')[:10]
 return render(request, 'admin/todo_list.html', {
 'todos': todos
 })
```

Now let's register these views in our **urls.py** file:

```
urlpatterns = [
 path('admin/', admin.site.urls),
 path('api/categories/', views.get_categories, name='get_categories'),
 path('api/get-category-dishes/', views.get_category_dishes, name='get-
category-dishes'),
 path('api/recent-orders/', views.recent_orders, name='recent-orders'),
 path('api/todo-widget/', views.todo_widget, name='todo-widget'),
 path('api/add-todo/', views.add_todo, name='add-todo'),
 path('api/complete-todo/<uuid:todo_id>/', views.complete_todo,
name='complete-todo'),
 path('silk/', include('silk.urls', namespace='silk')),
]
```

## Create TODO Widget Template

Our widget will be composed of two parts:

- A form to add a new TODO
- A list of TODOs

Let's create the template files:

```
touch backoffice/templates/admin/todo_widget.html
touch backoffice/templates/admin/todo_list.html
```

and first create the TODO widget template:

```
<div class="todo-widget">
 <h3>Quick TODOs</h3>

 <!-- Add TODO form -->
 <form hx-post="/api/add-todo/"
 hx-target="#todo-list"
 hx-swap="innerHTML"
 hx-on::after-request="this.reset()"
 class="add-todo-form">
```



```

 {% csrf_token %}
 <div class="form-row">
 <input type="text"
 name="title"
 placeholder="Add a new TODO..."
 class="todo-input"
 required>
 <select name="priority" class="priority-select">
 <option value="low">Low</option>
 <option value="medium" selected>Medium</option>
 <option value="high">High</option>
 </select>
 <button type="submit" class="add-btn">+</button>
 </div>
</form>

<!-- TODO list -->
<div id="todo-list">
 {% include "admin/todo_list.html" %}
</div>
</div>

<style>
 .todo-widget {
 margin-top: 20px;
 }

 .todo-widget h3 {
 margin-top: 0;
 color: #417690;
 border-bottom: 2px solid #e9ecef;
 padding-bottom: 10px;
 }

 .add-todo-form {
 margin-bottom: 15px;
 }

```

```

.form-row {
 display: flex;
 gap: 10px;
 align-items: center;
}

.todo-input {
 flex: 1;
 padding: 8px 12px;
 border: 1px solid #ddd;
 border-radius: 4px;
 font-size: 14px;
}

.priority-select {
 padding: 8px;
 border: 1px solid #ddd;
 border-radius: 4px;
 font-size: 14px;
 background: white;
}

.add-btn {
 width: 40px;
 height: 36px;
 background: #417690;
 color: white;
 border: none;
 border-radius: 4px;
 cursor: pointer;
 font-size: 18px;
 font-weight: bold;
}

.add-btn:hover {
 background: #2c5766;
}
</style>

```

and now the list template:

```
{% if todos %}
 <div class="todos">
 {% for todo in todos %}
 <div class="todo-item priority-{{ todo.priority }}">
 <div class="todo-content">
 {{ todo.title }}
 {{ todo.get_priority_
 display }}
 </div>
 <form hx-post="/api/complete-todo/{{ todo.id }}"
 hx-target="#todo-list"
 hx-swap="innerHTML"
 style="display: inline;">
 {% csrf_token %}
 <button type="submit" class="complete-btn">✓</button>
 </form>
 </div>
 {% endfor %}
 </div>
{% else %}
 <p class="no-todos">No pending TODOs</p>
{% endif %}

<style>
 .todos {
 max-height: 300px;
 overflow-y: auto;
 }

 .todo-item {
 display: flex;
 justify-content: space-between;
 align-items: center;
 padding: 10px;
 margin-bottom: 8px;
 }
```

```
 border: 1px solid #e9ecef;
 border-radius: 4px;
 background: #f8f9fa;
 border-left: 4px solid #ddd;
}

.todo-item.priority-high {
 border-left-color: #dc3545;
}

.todo-item.priority-medium {
 border-left-color: #ffc107;
}

.todo-item.priority-low {
 border-left-color: #28a745;
}

.todo-content {
 flex: 1;
 display: flex;
 flex-direction: column;
 gap: 4px;
}

.todo-title {
 font-weight: 500;
 color: #333;
}

.todo-priority {
 font-size: 12px;
 color: #666;
 text-transform: uppercase;
}

.complete-btn {
 width: 30px;
 height: 30px;
 background: #28a745;
```

```

 color: white;
 border: none;
 border-radius: 50%;
 cursor: pointer;
 font-size: 14px;
 display: flex;
 align-items: center;
 justify-content: center;
}

.complete-btn:hover {
 background: #218838;
}

.no-todos {
 text-align: center;
 color: #6c757d;
 font-style: italic;
 padding: 20px 0;
}
</style>

```

---

Please notice that to avoid CSRF issue with HTMX when calling the complete views, we embed the submit button and CSRF token in a form.

---

## Integrate TODO Widget into Admin Index

Let's integrate the widget (see Figure 19-3) to our `index.html` file, as we have done previously with the recent orders widget:

```

<!-- Second column: Existing dashboard -->
<div class="dashboard-column">
 <div class="dashboard-stats">
 <div class="stat-card">
 <h3>{% total_restaurants %}</h3>
 <p>Total Restaurants</p>
 </div>
 </div>
</div>

```

```

</div>
<!-- Using inclusion tag for complex widgets -->
{% dashboard_statistics %}
</div>

<!-- TODO Widget -->
<div hx-get="/api/todo-widget/" hx-trigger="load">
 Loading TODOs...
</div>

{% include "admin/app_list.html" with app_list=app_list show_
changelinks=True %}
</div>

```

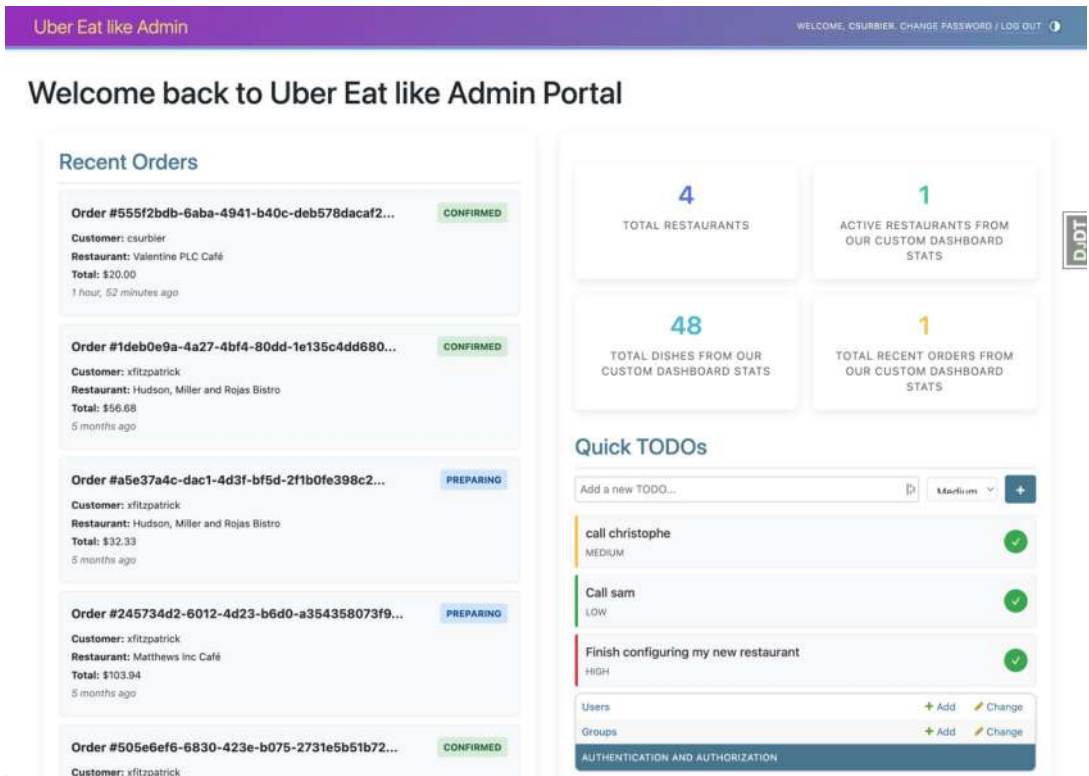


**Figure 19-3.** TODO widget with HTMX after completing a task (list refreshes without full page reload)

If we click the green checkmark, it will call the complete TODO view and mark the task as completed. Then the list will refresh without the task marked as completed.

CHAPTER 20

# Charts and Analytics

Another common feature of admin interfaces is the ability to display charts and analytics. This can help users understand trends and patterns in their data (see Figure 20-1).

For example, we will modify the order change list template to display a chart with orders by status and orders over time.

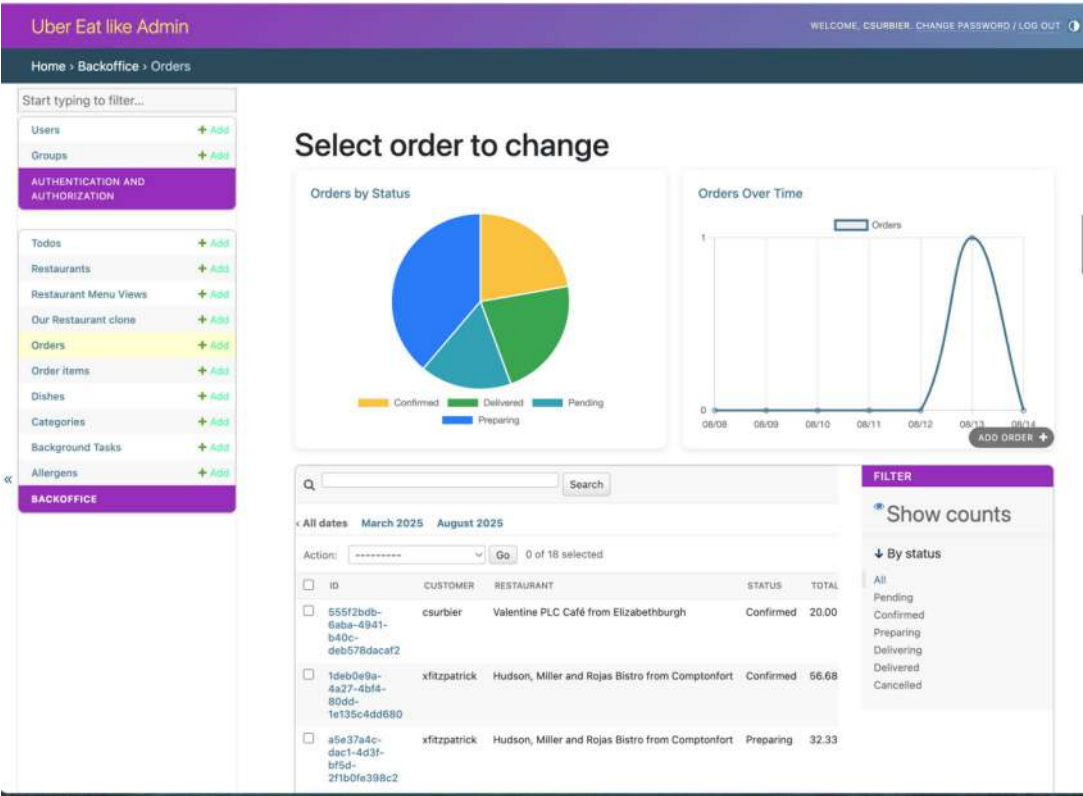


**Figure 20-1.** Order changelist with Chart.js analytics (orders by status and over time)

Let's go. First, we need to create a specific folder **order** in our backoffice admin templates folder and then copy the default Django Admin `change_list.html` template:

```
mkdir backoffice/templates/admin/backoffice/order
cp ../venv/lib/python3.12/site-packages/django/contrib/admin/templates/
admin/change_list.html backoffice/templates/admin/backoffice/order/
```

Then we will modify the template to include Chart.js integration and will use the CDN version of it.

## Add Chart.js Integration to Order Change List Template

```
{% block extrahead %}
 {{ block.super }}
 {{ media.js }}
 <script src="{% static 'admin/js/filters.js' %}" defer></script>
 <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
{% endblock %}
```

Next, we will modify the block content section to add two divs that will display the chart data:

```
{% block content %}
 <div id="content-main">
 <!-- Order Charts Section -->
 <div class="order-charts-section" style="margin-bottom: 20px;">
 <div class="charts-container" style="display: grid; grid-template-
 columns: 1fr 1fr; gap: 20px; margin-bottom: 20px;">
 <div class="chart-card">
 <h3>Orders by Status</h3>
 <canvas id="ordersStatusChart" width="400"
 height="200"></canvas>
 </div>
 <div class="chart-card">
 <h3>Orders Over Time</h3>
```



```

 <canvas id="ordersTimeChart" width="400" height="200"></canvas>
 </div>
 </div>
 </div>

```

and will need to style these charts:

```

<style>
 .chart-card {
 background: white;
 border-radius: 8px;
 box-shadow: 0 2px 8px rgba(0, 0, 0, 0.1);
 padding: 20px;
 height: 350px;
 display: flex;
 flex-direction: column;
 }

 .chart-card h3 {
 margin-top: 0;
 margin-bottom: 15px;
 color: #417690;
 font-size: 16px;
 flex-shrink: 0;
 }

 .chart-card canvas {
 flex: 1;
 height: 0 !important;
 max-height: 280px;
 }

 @media (max-width: 768px) {
 .charts-container {
 grid-template-columns: 1fr !important;
 }
 }
</style>

```

To get data, we will call an API endpoint that will return the data for the charts. So, we need to add a JavaScript to handle this:

```
<script>
 document.addEventListener('DOMContentLoaded', function() {
 // Fetch order chart data
 fetch('/api/order-charts/')
 .then(response => response.json())
 .then(data => {
 // Orders by Status Chart (Pie Chart)
 const statusCtx = document.getElementById('ordersStatusChart').getContext('2d');
 new Chart(statusCtx, {
 type: 'pie',
 data: {
 labels: data.status_labels,
 datasets: [{
 data: data.status_counts,
 backgroundColor: [
 '#ffc107', // pending
 '#28a745', // confirmed
 '#17a2b8', // preparing
 '#007bff', // delivering
 '#6f42c1', // delivered
 '#dc3545' // cancelled
]
 }]
 },
 options: {
 responsive: true,
 maintainAspectRatio: false,
 aspectRatio: 1,
 plugins: {
 legend: {
 position: 'bottom'
 }
 }
 }
 })
 })
 })
</script>
```

```

 }
 });

 // Orders Over Time Chart (Line Chart)
 const timeCtx = document.getElementById('ordersTimeChart').
 getContext('2d');
 new Chart(timeCtx, {
 type: 'line',
 data: {
 labels: data.time_labels,
 datasets: [{
 label: 'Orders',
 data: data.time_counts,
 borderColor: '#417690',
 backgroundColor: 'rgba(65, 118, 144, 0.1)',
 tension: 0.4
 }]
 },
 options: {
 responsive: true,
 maintainAspectRatio: false,
 aspectRatio: 2,
 scales: {
 y: {
 beginAtZero: true,
 ticks: {
 stepSize: 1
 }
 }
 }
 }
 });
})
.catch(error => console.error('Error loading chart data:', error));
});
</script>

```

## Create View for Order Chart Data

As mentioned previously, we need a new endpoint/view to handle the chart data request:

```
from django.db.models import Count
from datetime import datetime, timedelta

...
@staff_member_required
def order_charts_data(request):
 """
 API endpoint to provide chart data for orders
 """

 # Orders by Status
 status_data = Order.objects.values('status').
 annotate(count=Count('id')).order_by('status')
 status_labels = [item['status'].title() for item in status_data]
 status_counts = [item['count'] for item in status_data]

 # Orders over last 7 days
 end_date = timezone.now().date()
 start_date = end_date - timedelta(days=6)

 time_data = []
 time_labels = []

 for i in range(7):
 date = start_date + timedelta(days=i)
 count = Order.objects.filter(created_at__date=date).count()
 time_data.append(count)
 time_labels.append(date.strftime('%m/%d'))

 return JsonResponse({
 'status_labels': status_labels,
 'status_counts': status_counts,
 'time_labels': time_labels,
 'time_counts': time_data
 })
```

and add the new endpoint called **api/order-charts/** to our **urls.py** file:

```
path('api/order-charts/', views.order_charts_data, name='order-charts'),
```

That's it; we have integrated the order chart data into our Django Admin.

## Alternative: Build the Same Charts with **json\_script** (No Extra API Call)

The `fetch('/api/order-charts/')` approach is great when you want to reuse the same data outside the admin (or refresh it without reloading the page). But for a simple changelist dashboard, we can also compute the series on the server and pass them to JavaScript **safely** using Django's built-in **json\_script** filter.

This pattern

- Avoids an extra HTTP round trip (the data is already available when we render the page)
- Avoids hard-coding URLs in your JavaScript
- Avoids ad hoc JSON string building inside templates (safer against XSS)

### 1. Build the Chart Payload As a Python dict

Create a helper that returns a JSON-serializable dictionary (same structure as our API response):

```
from django.db.models import Count
from django.utils import timezone
from datetime import timedelta
from .models import Order
```

```
def build_order_charts_data():
 # Orders by Status
 status_data = Order.objects.values('status').
 annotate(count=Count('id')).order_by('status')
 status_labels = [item['status'].title() for item in status_data]
 status_counts = [item['count'] for item in status_data]
```

```

Orders over last 7 days
end_date = timezone.now().date()
start_date = end_date - timedelta(days=6)

time_data = []
time_labels = []
for i in range(7):
 date = start_date + timedelta(days=i)
 count = Order.objects.filter(created_at__date=date).count()
 time_data.append(count)
 time_labels.append(date.strftime('%m/%d'))

return {
 'status_labels': status_labels,
 'status_counts': status_counts,
 'time_labels': time_labels,
 'time_counts': time_data,
}

```

## 2. Add the Payload to the changelist Context

Override `changelist_view()` in `OrderAdmin` and pass the dict as `order_charts`:

```

from import_export.admin import ImportExportModelAdmin
from backoffice.views import build_order_charts_data

@admin.register(Order)
class OrderAdmin(ImportExportModelAdmin):
 # ... your existing OrderAdmin configuration ...
 change_list_template = 'admin/backoffice/order/change_list.html'

 def changelist_view(self, request, extra_context=None):
 extra_context = extra_context or {}
 extra_context['order_charts'] = build_order_charts_data()
 return super().changelist_view(request, extra_context)

```

### 3. Embed the JSON in the Template with `json_script`, then Parse It

In `admin/backoffice/order/change_list.html`, output the JSON payload near the top of `{% block content %}`:

```
{{ order_charts|json_script:"order-charts-data" }}
```

Then replace the `fetch(...)` block with a parse step:

**<script>**

```
document.addEventListener('DOMContentLoaded', function() {
 const data = JSON.parse(document.getElementById('order-charts-data').
 textContent);

 // Now use data.status_labels / data.status_counts / data.time_labels
 / data.time_counts
 // ... (same Chart.js initialization code as before) ...
});
```

**</script>**

---

If you keep the JSON API endpoint, you get both options: `json_script` for fast server-rendered admin pages and `/api/order-charts/` for external clients or future asynchronous refresh.

---

## Report Generation

### CSV Export

In the previous chapter, we have seen how to manually create a CSV export of our restaurant data. Exporting data or importing data can be very useful in an admin. Fortunately, there is a Django package to do this easily called [django import export](https://django-import-export.readthedocs.io/en/latest/) (<https://django-import-export.readthedocs.io/en/latest/>):

```
pip install django-import-export
```

And we add it to our `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
 ...
 'import_export',
)
```

Then, to use it on orders (see new import or export buttons on Figure 20-2), we need to change the admin class to the import export one: **ImportExportModelAdmin**. For every model that we need to export, we need to add the **ImportExportModelAdmin** to the admin class:

```
from import_export.admin import ImportExportModelAdmin

@admin.register(Order)
class OrderAdmin(ImportExportModelAdmin):
 list_display = ('id', 'customer', 'restaurant', 'status', 'total_
amount', 'created_at')
 list_filter = ('status',)
 search_fields = ('customer__username', 'restaurant__name')
 list_select_related = ('customer', 'restaurant',)
 date_hierarchy = 'created_at'
 change_list_template = 'admin/backoffice/order/change_list.html'
```

---

We also need to add the **change\_list\_template** to our **OrderAdmin** class to use the custom template for our order list and not the **django-import-export** template.

---



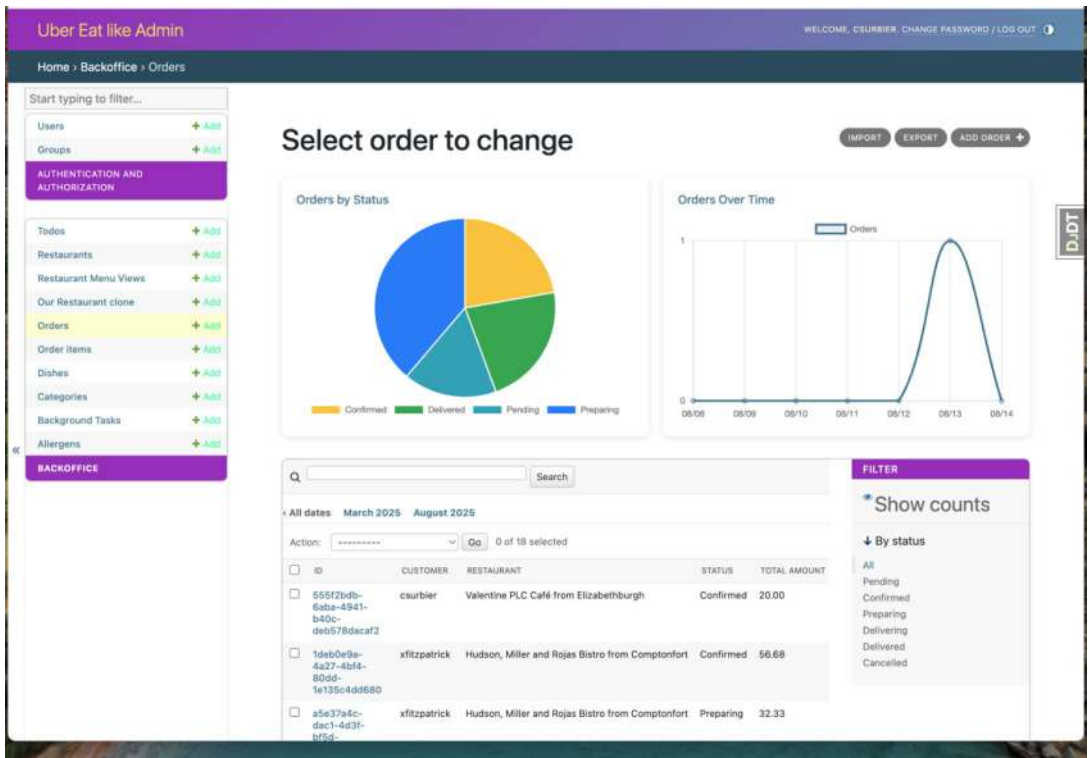


**Figure 20-2.** Order changelist with Django-import-export actions alongside charts

To avoid the export and import button to collapse with our charts, we need to modify a little bit our **change\_list.html** template and move the **object\_tools** block before the charts section. We also add some margin to the charts section to avoid the charts to be too close to the export and import button.

```
{% block content %}
<div id="content-main">
 {% block object-tools %}
 <ul class="object-tools">
 {% block object-tools-items %}
 {% change_list_object_tools %}
 {% endblock %}

 {% endblock %}
</div>
```

```

{% endblock %}

<!-- Order Charts Section -->
<div class="order-charts-section" style="margin: 20px 0 30px 0;
clear: both;">
 <div class="charts-container" style="display: grid; grid-template-
columns: 1fr 1fr; gap: 20px;">
 <div class="chart-card">
 <h3>Orders by Status</h3>
 <canvas id="ordersStatusChart" width="400"
height="200"></canvas>
 </div>
 <div class="chart-card">
 <h3>Orders Over Time</h3>
 <canvas id="ordersTimeChart" width="400"
height="200"></canvas>
 </div>
 </div>
</div>

```

- To export data, just select order items in the list and click the export button; it will automatically go to a new screen on which you will be able to select the fields you want to export and the form at to export to (see Figure 20-3).

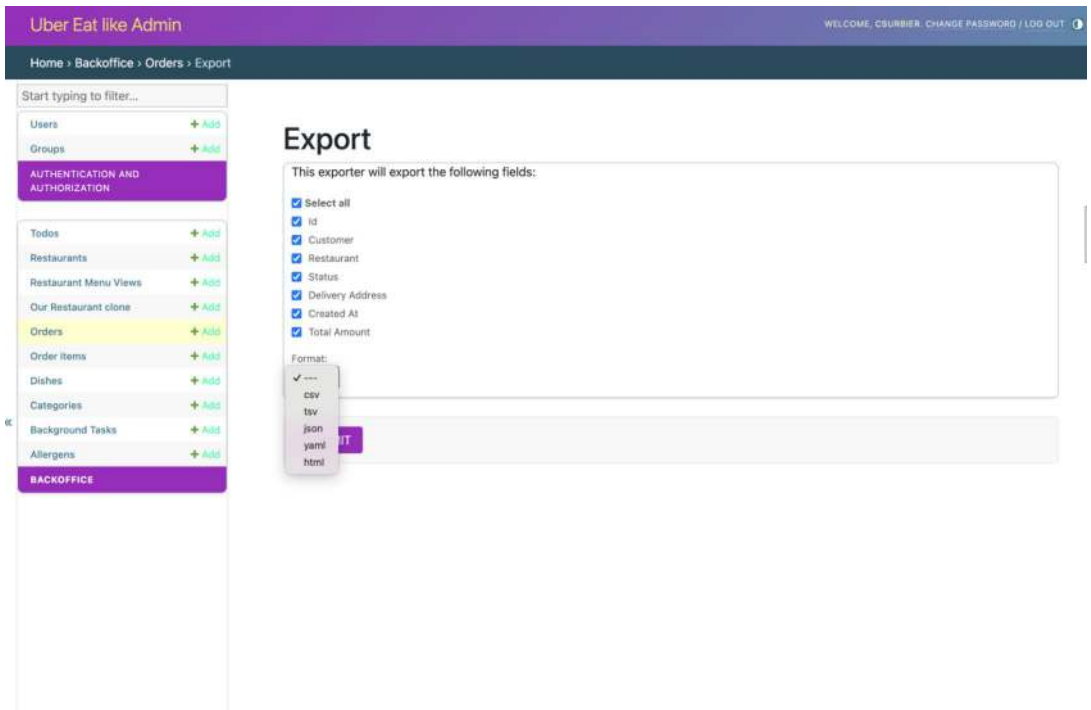


**Figure 20-3.** Export workflow: selecting fields and format for order data

- To import data (see Figure 20-4), click the import button and you will be able to select the file to import and the format to import from.

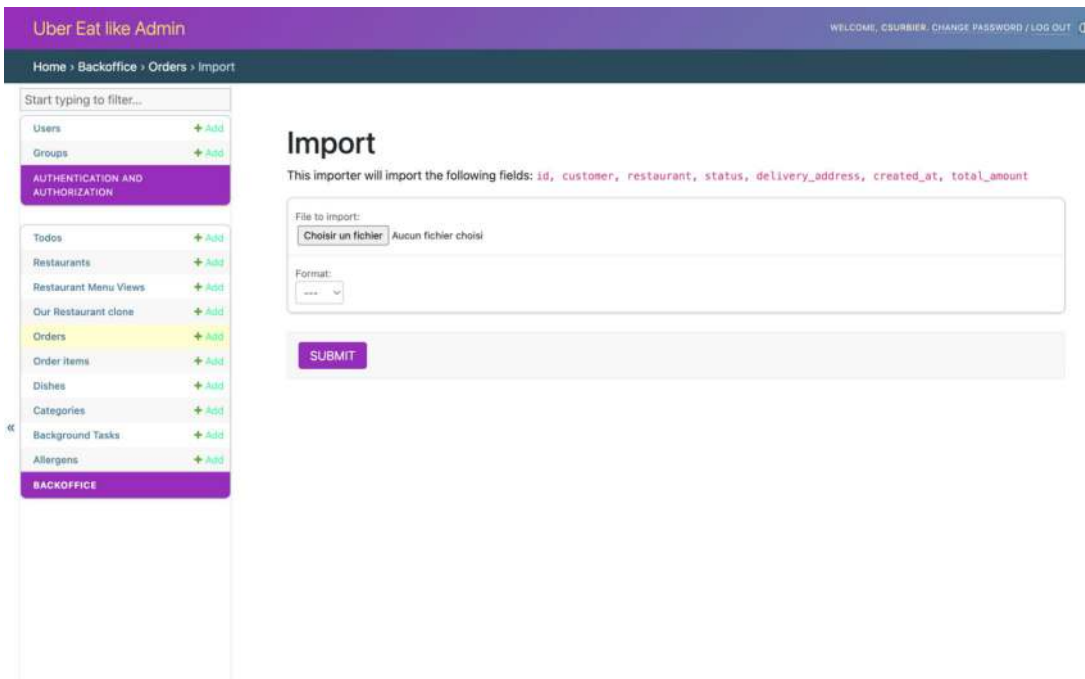


**Figure 20-4.** *Import workflow: uploading a file and choosing import options*

This plugin is really useful, and to know more about its possibilities, don't hesitate to refer to the [documentation](https://django-import-export.readthedocs.io/en/latest/advanced_usage.html) ([https://django-import-export.readthedocs.io/en/latest/advanced\\_usage.html](https://django-import-export.readthedocs.io/en/latest/advanced_usage.html)).

## PDF Reports

It's also possible to create PDF reports and not just CSV. To do so, we will use the **reportlab** package:

```
pip install reportlab==4.4.3
```

In this section, you will learn how to

- Return a PDF from a Django view (HttpResponse with application/pdf)
- Build a PDF document with ReportLab (SimpleDocTemplate + a story list)

- Add **tables** and a simple **chart**
- Wire the PDF export into Django Admin with a link in the changelist **object tools**

## Imports and Optional Dependency Guard

ReportLab is an optional dependency for our project. A simple pattern is to import it in a try/except block and return a helpful message when it is not installed:

```
from django.http import HttpResponse
from django.contrib.admin.views.decorators import staff_member_required
from django.utils import timezone
from django.db.models import Avg, Count, Sum

from backoffice.models import Order

try:
 from reportlab.lib import colors
 from reportlab.lib.pagesizes import A4
 from reportlab.platypus import SimpleDocTemplate, Table, TableStyle,
 Paragraph, Spacer
 from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
 from reportlab.lib.units import inch
 from reportlab.graphics.shapes import Drawing
 from reportlab.graphics.charts.piecharts import Pie
 REPORTLAB_AVAILABLE = True
except ImportError:
 REPORTLAB_AVAILABLE = False
```

## PDF Helper: A Simple Pie Chart

We start with a small helper that returns a ReportLab Drawing containing a pie chart:

```
def create_pie_chart_drawing(data, labels, title):
 """Helper function to create a pie chart for PDF"""
 drawing = Drawing(300, 250)
 pie = Pie()
```

```

pie.x = 50
pie.y = 25 # Lower position to create space at top
pie.width = 200
pie.height = 200
pie.data = data
pie.labels = labels
pie.slices.strokeWidth = 0.5
drawing.add(pie)
return drawing

```

```
@staff_member_required
```

```
def generate_orders_pdf_report(request):
```

```
 """
```

```
 Generate a comprehensive PDF report for orders with charts and tables
 """
```

```
if not REPORTLAB_AVAILABLE:
```

```
 return HttpResponse("ReportLab not installed. Please install with:
 pip install reportlab",
 content_type="text/plain")

```

```
Create HTTP response with PDF content type
```

```
response = HttpResponse(content_type='application/pdf')
response['Content-Disposition'] = 'attachment; filename="orders_
report.pdf"'

```

```
Create PDF document
```

```
doc = SimpleDocTemplate(response, pagesize=A4, topMargin=0.5*inch)
```

```
Get styles
```

```
styles = getSampleStyleSheet()
title_style = ParagraphStyle(
 'CustomTitle',
 parent=styles['Heading1'],
 fontSize=18,
 spaceAfter=30,
 textColor=colors.HexColor('#417690')
)

```

```

Build PDF content
story = []

Title
story.append(Paragraph("Orders Report", title_style))
story.append(Spacer(1, 12))

Executive Summary
total_orders = Order.objects.count()
total_revenue = Order.objects.aggregate(total=Sum('total_amount'))
['total'] or 0
pending_orders = Order.objects.filter(status='pending').count()

summary_data = [
 ['Metric', 'Value'],
 ['Total Orders', str(total_orders)],
 ['Total Revenue', f'${total_revenue:.2f}'],
 ['Pending Orders', str(pending_orders)],
 ['Report Generated', timezone.now().strftime('%Y-%m-%d %H:%M')]
]

summary_table = Table(summary_data, colWidths=[2*inch, 2*inch])
summary_table.setStyle(TableStyle([
 ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#417690')),
 ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
 ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
 ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
 ('FONTSIZE', (0, 0), (-1, 0), 12),
 ('BOTTOMPADDING', (0, 0), (-1, 0), 12),
 ('BACKGROUND', (0, 1), (-1, -1), colors.beige),
 ('GRID', (0, 0), (-1, -1), 1, colors.black),
]))

story.append(Paragraph("Executive Summary", styles['Heading2']))
story.append(summary_table)
story.append(Spacer(1, 20))

Orders by Status Chart

```

```

if total_orders > 0:
 status_data = Order.objects.values('status').
 annotate(count=Count('id'))
 chart_data = [item['count'] for item in status_data]
 chart_labels = [item['status'].title() for item in status_data]

 story.append(Paragraph("Orders by Status", styles['Heading2']))
 story.append(Spacer(1, 12))
 story.append(create_pie_chart_drawing(chart_data, chart_labels,
 "Orders by Status"))
 story.append(Spacer(1, 20))

Recent Orders Table
story.append(Paragraph("Recent Orders (Last 20)", styles['Heading2']))

recent_orders = Order.objects.select_related('customer', 'restaurant').
order_by('-created_at')[:20]

if recent_orders:
 table_data = [['Order ID', 'Customer', 'Restaurant', 'Status',
 'Total', 'Date']]

 for order in recent_orders:
 table_data.append([
 str(order.id)[:8] + '...',
 order.customer.username,
 order.restaurant.name[:20] + ('...' if len(order.
 restaurant.name) > 20 else ''),
 order.get_status_display(),
 f'${order.total_amount}',
 order.created_at.strftime('%m/%d/%Y')
])

orders_table = Table(table_data, colWidths=[1.2*inch, 1*inch,
1.5*inch, 1*inch, 0.8*inch, 1*inch])
orders_table.setStyle(TableStyle([
 ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#417690')),
 ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),

```



```

 ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
 ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
 ('FONTSIZE', (0, 0), (-1, 0), 10),
 ('FONTSIZE', (0, 1), (-1, -1), 8),
 ('BOTTOMPADDING', (0, 0), (-1, 0), 12),
 ('BACKGROUND', (0, 1), (-1, -1), colors.lightgrey),
 ('GRID', (0, 0), (-1, -1), 1, colors.black),
 ('VALIGN', (0, 0), (-1, -1), 'MIDDLE'),
]))

 story.append(orders_table)
else:
 story.append(Paragraph("No recent orders found.",
 styles['Normal']))

 story.append(Spacer(1, 20))

Orders by Restaurant
 story.append(Paragraph("Orders by Restaurant", styles['Heading2']))
 restaurant_stats = Order.objects.values('restaurant__name').annotate(
 count=Count('id'),
 revenue=Sum('total_amount'),
 avg_order_value=Avg('total_amount')
).order_by('-count')[:10]

if restaurant_stats:
 restaurant_data = [['Restaurant', 'Orders', 'Revenue', 'Avg. Order
Value']]
 for stat in restaurant_stats:
 avg_value = stat.get('avg_order_value') or 0
 revenue = stat.get('revenue') or 0
 restaurant_data.append([
 stat['restaurant__name'][:25] + ('...' if
 len(stat['restaurant__name']) > 25 else ''),
 str(stat['count']),
 f'${revenue:.2f}',
 f'${avg_value:.2f}'
])

```

```

restaurant_table = Table(restaurant_data, colWidths=[2.6*inch,
0.9*inch, 1.2*inch, 1.3*inch])
restaurant_table.setStyle(TableStyle([
 ('BACKGROUND', (0, 0), (-1, 0), colors.HexColor('#417690')),
 ('TEXTCOLOR', (0, 0), (-1, 0), colors.whitesmoke),
 ('ALIGN', (0, 0), (-1, -1), 'CENTER'),
 ('FONTNAME', (0, 0), (-1, 0), 'Helvetica-Bold'),
 ('FONTSIZE', (0, 0), (-1, 0), 10),
 ('FONTSIZE', (0, 1), (-1, -1), 9),
 ('BOTTOMPADDING', (0, 0), (-1, 0), 12),
 ('BACKGROUND', (0, 1), (-1, -1), colors.lightblue),
 ('GRID', (0, 0), (-1, -1), 1, colors.black),
]))

story.append(restaurant_table)

Footer
story.append(Spacer(1, 30))
footer_style = ParagraphStyle(
 'Footer',
 parent=styles['Normal'],
 fontSize=10,
 textColor=colors.grey,
 alignment=1 # Center alignment
)
story.append(Paragraph("Generated by Django Admin - Restaurant
Management System", footer_style))

Build PDF
doc.build(story)
return response

```

Add a new endpoint to get the data:

```

path('api/orders-pdf-report/', views.generate_orders_pdf_report,
name='orders-pdf-report')

```

and then modify our `order_change_list` template to include a new button, **Generate PDF Report** (see Figure 20-5 and Figure 20-6 for PDF results), which will call the endpoint:

```
{% block content %}
<div id="content-main">
 {% block object-tools %}
 <ul class="object-tools">
 {% block object-tools-items %}
 {% change_list_object_tools %}

 <a href="/api/orders-pdf-report/" class="addlink"
 target="_blank">
  Generate PDF Report

 {% endblock %}

 {% endblock %}
{% endblock %}
```

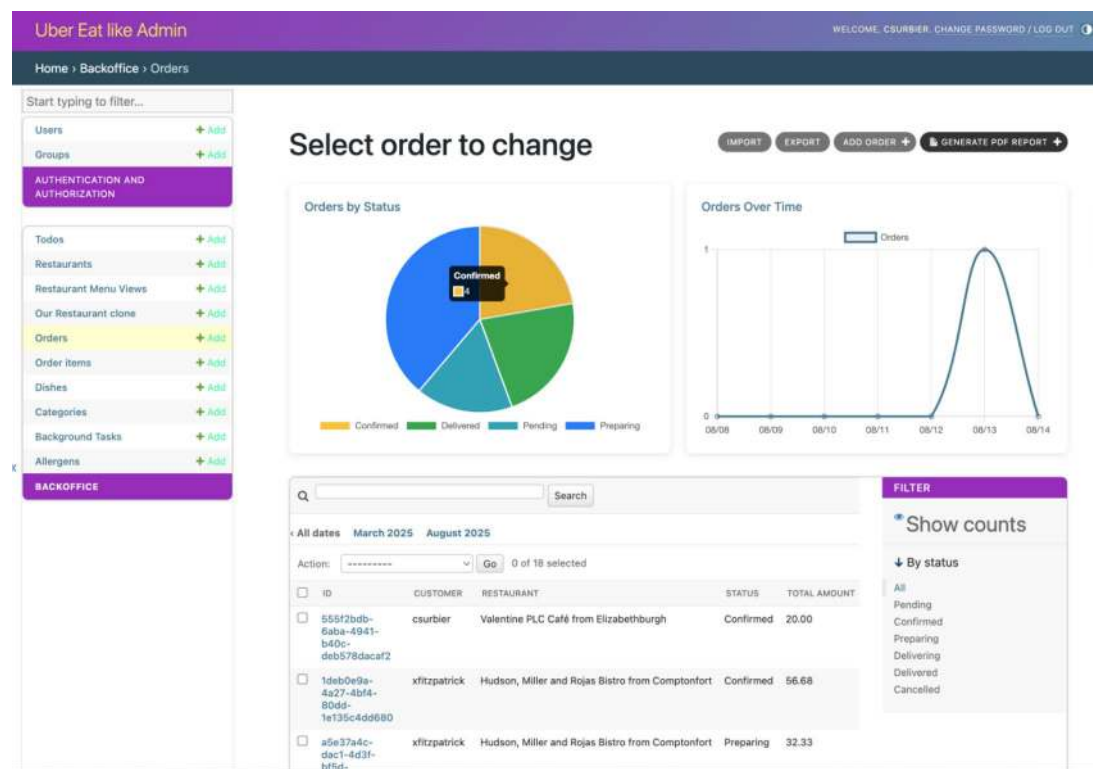


**Figure 20-5.** Custom “Generate PDF Report” link added to the order changelist object tools

The view returns a PDF file with the report so the browser will download it.

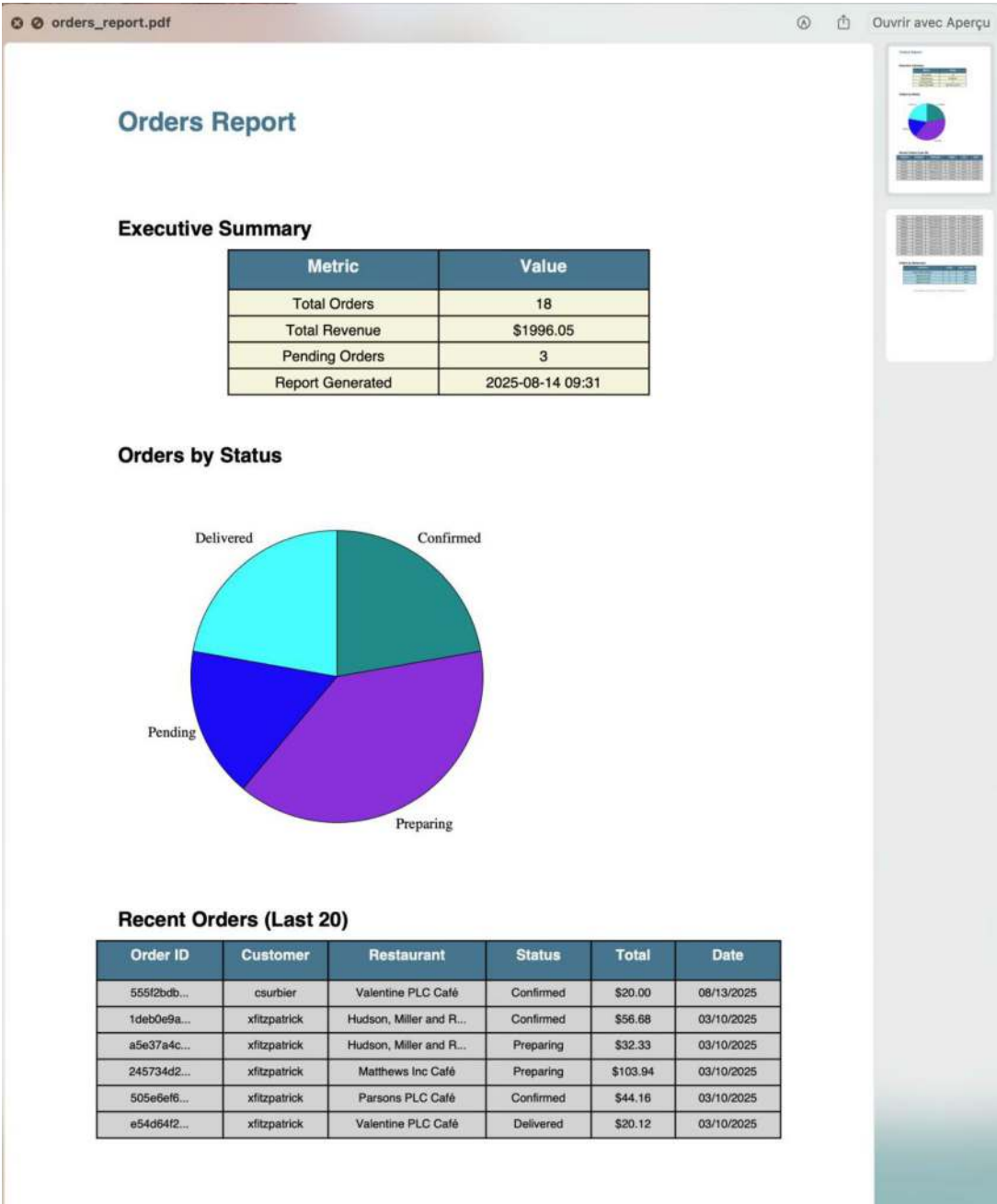


**Figure 20-6.** Generated PDF report opened in the browser (order summary and charts)

## Avoiding CDNs: Serving Frontend Assets with Django staticfiles

So far, we used a CDN for Chart.js. This is convenient for demos, but in many environments (internal networks, offline deployments, strict security policies, CSP), **CDNs are not an option.**

Django already ships with the `staticfiles` app, which can collect and serve your static assets during development, and prepares them for production deployment via `collectstatic`.

### Put the Asset in Your Project

The simplest approach is to vendor the library in your app's `static/` folder. For example:

```
backoffice/static/backoffice/vendor/chartjs/chart.umd.min.js
```

You can download Chart.js and copy it there (any method is fine: download from the release page, your package manager, etc.). The key point is that **the file lives under an app `static/` directory** so Django can find it.

### Reference It in Templates with `{% static %}`

Replace the CDN tag:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

with a local static reference:

```
{% load static %}
<script src="{% static 'backoffice/vendor/chartjs/chart.umd.min.js' %}"></script>
```

This works because `django.contrib.staticfiles` knows how to locate files under each app's `static/` directory and generates the correct URL.

## Settings and Deployment Notes

In `settings.py`, `STATIC_URL` defines the URL prefix where static assets live:

```
STATIC_URL = "static/"
```

For production deployments, you typically also set a `STATIC_ROOT` (where `collectstatic` gathers all assets) and optionally `STATICFILES_DIRS` for project-level static folders:

```
STATIC_ROOT = BASE_DIR / "staticfiles"
STATICFILES_DIRS = [BASE_DIR / "static"]
```

Then run

```
python manage.py collectstatic
```

In production, your web server (or storage/CDN you control) serves files from `STATIC_ROOT`. During development, `runserver` serves static files automatically when `DEBUG=True`.

## Summary

In this chapter, we have learned how to transform the Django Admin interface into a powerful, modern dashboard that goes far beyond basic CRUD operations. We started by mastering Django's template inheritance system, understanding how to extend base templates and override specific blocks to customize the admin interface while preserving its core functionality.

We explored advanced customization techniques that turn the admin into a comprehensive business management tool, creating a responsive two-column layout for the admin index page, demonstrating how to override Django's default templates while maintaining compatibility with existing functionality. We then integrated HTMX to add real-time components, including an auto-refreshing recent orders display that updates every ten seconds without requiring page reloads.

We built a complete TODO management widget that showcases HTMX's capabilities for seamless user interactions, including instant form submissions and dynamic content updates. This widget demonstrated how to create custom models, views, and templates that integrate smoothly with Django Admin's existing architecture.

Our exploration of data visualization covered two powerful approaches: Chart.js for interactive web charts and ReportLab for professional PDF reports. We created dynamic charts showing order statistics and trends, then learned how to generate comprehensive PDF reports that combine data tables, visual charts, and professional formatting suitable for executive presentations.

Throughout the chapter, we emphasized responsive design principles, ensuring our customizations work seamlessly across desktop and mobile devices.

The techniques learned in this chapter enable developers to create sophisticated admin interfaces that rival custom-built dashboards. These skills are essential for building production-ready applications that provide both technical functionality and outstanding user experience.



# PART V

# Security and Access Control

In this final part, we will focus on enhancing the security and access control of our Django Admin interface: covering access protection, permission and role-based systems, audit logging, and general security best practices.

## CHAPTER 21

# Protect Django Admin Access

As we learned, the admin interface offers extensive access to our database, user accounts, and application settings.

By default, Django Admin is accessible to anyone who knows the URL (which is by default **/admin**), making it a prime target for attackers. A compromised admin account can lead to devastating consequences: data breaches, unauthorized content modifications, user account manipulation, and complete system takeover.

This is why it is crucial to implement robust security measures to protect the Django Admin interface, such as

- Customizing the admin URL
- Implementing IP blocking
- Rate limiting
- Always use HTTPS and SSL/TLS encryption
- Implementing two-factor authentication

## Customizing the Admin URL

By default, Django Admin is accessible at **/admin**. The first simple thing we should do is customizing the admin URL like adding a random string to our admin URL (or set a completely different one), which will result in a 404 Not Found page if someone tries to access the default **/admin** url (see Figure 21-1):

```
urlpatterns = [
 path('admine44fjgjdfjgfhjgf', admin.site.urls),
```

or

```
urlpatterns = [
 path('mycustomadminurl', admin.site.urls),
]
```

---

Even if there is a way to find URLs (for advanced attackers), it's an easy step to obfuscate them by adding a random string to the URL. **An important thing is not to forget to set `DEBUG` to `False` in production.** Otherwise, performance will be degraded and also security will be compromised since an attacker will be able to see very easily all available endpoints because of the 404 page.

---



**Figure 21-1.** Django debug 404 page listing available URL patterns (keep `DEBUG` off in production)

An even better solution is to set the admin URL in an environment variable and use a different one for each environment (local, staging, production):

```
urlpatterns = [
 path(settings.ADMIN_URL, admin.site.urls),
]
```

and in settings.py:

```
ADMIN_URL = os.getenv('ADMIN_URL', 'mycustomadminurl')
```

With this approach, we can change the admin URL in production without changing the code, on regular basis.

## IP Blocking/Filtering

If your admin is accessible from the internet, you should block all IP addresses except those of your own. We can use `django-ipware` (<https://github.com/un33k/django-ipware>) to get the client's IP address and block it.

```
pip install django-ipware
```

Then we will create a middleware to block all IP addresses except those of our own:

```
touch backoffice/ipmiddleware.py
import logging

from django.conf import settings
from django.shortcuts import redirect, reverse
from ipware import get_client_ip

logger = logging.getLogger(__name__)

class IPAdminMiddleware:
 """
 Middleware to restrict access to the admin panel by IP address.
 """

 def __init__(self, get_response):
 self.get_response = get_response

 def __call__(self, request):
 if request.path.startswith(f"/{settings.ADMIN_URL}"):
 print("Here we are")
 client_ip, is_routable = get_client_ip(request)
 print(f"Client IP: {client_ip}")
```

```

 if client_ip not in settings.WHITELISTED_IPS:
 print(
 "A user tried to access the admin panel from an
 unauthorized IP address."
)
 return redirect("https://www.google.com")
 response = self.get_response(request)
 return response

```

Now add it in settings.py:

```

MIDDLEWARE = [
 ...
 'backoffice.ipmiddleware.IPAdminMiddleware',
]
WHITELISTED_IPS = os.getenv("WHITELISTED_IPS", "127.0.0.2").split(",")

```

If we try to access the admin panel as usual using the URL **http://127.0.0.1:8000/mycustomadminurl/**, we will be redirected to Google because our IP address (by default our local IP address is 127.0.0.1) is not in the whitelist. Indeed, with the middleware we have set, only the IP address 127.0.0.2 will be able to access the admin panel.

You can see the logs in the console:

Here we are

Client IP: 127.0.0.1

A user tried to access the admin panel from an unauthorized IP address.

---

To test the redirection, I set 127.0.0.2; if you want to allow access from your local machine, you can set it back to **127.0.0.1**. If you have access to the network or load balancer configuration, you can also ask the network team to set the whitelist to the load balancer IP address.

---

## Brute-Force Protection (Login Lockout)

We can use `django-axes` (<https://github.com/jazzband/django-axes>) to protect the admin login against brute-force attacks by tracking repeated failed authentication attempts and temporarily locking out the source as shown on Figure 21-2 (e.g., by IP address, username, or a combination).

```
pip install django-axes
```

Add it to the `INSTALLED_APPS` in `settings.py`:

```
INSTALLED_APPS = [
 ...
 'axes',
]
```

Add the middleware in `settings.py`:

```
MIDDLEWARE = [
 ...
 'axes.middleware.AxesMiddleware',
]
```

And we also need to set the rules and the authentication backend:

```
Axes settings
AXES_FAILURE_LIMIT = 3
AXES_COOLOFF_TIME = 1 # hours
AXES_LOCKOUT_PARAMETERS = ["ip_address"]
AXES_RESET_ON_SUCCESS = True

AUTHENTICATION_BACKENDS = [
 'axes.backends.AxesBackend',
 'django.contrib.auth.backends.ModelBackend',
]
```

Run the migrations:

```
python manage.py migrate axes
```

Now log out from the admin panel and try to log in again but with a wrong password three times. You will be locked out for one hour.

AXES: Repeated login failure by {username: "\*\*\*\*\*", ip\_address: "\*\*\*\*\*", user\_agent: "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/131.0.0.0 Safari/537.36", path\_info: "/admin/login/"}. Updated existing record in the database.



**Figure 21-2.** Django-axes logout after repeated failed admin login attempts

---

If you want to unlock the admin panel, you can run the following command:

```
python manage.py axes_reset
```

---

## HTTPS and SSL/TLS Encryption

Even if it's quite obvious, we should always use HTTPS and SSL/TLS encryption to protect our admin panel: credentials and session cookies must never transit over plain HTTP.

In production, the HTTP → HTTPS redirect and TLS termination should ideally be enforced at the edge (load balancer/reverse proxy/web server such as nginx or Apache) so there is no “accidental HTTP” entry point.

On the Django side, prefer the built-in `SecurityMiddleware` settings rather than third-party packages (e.g., `django-sslify` is no longer maintained):

```
settings.py
SECURE_SSL_REDIRECT = True

If you're behind a reverse proxy/load balancer that sets
X-Forwarded-Proto
SECURE_PROXY_SSL_HEADER = ("HTTP_X_FORWARDED_PROTO", "https")
```

## Two-Factor Authentication

Two-factor authentication (2FA) is a security mechanism that requires users to provide two forms of authentication to access their accounts (see Figure 21-3). This adds an extra layer of security to the login process, making it more difficult for unauthorized users to access sensitive information.

We can use packages such as `django-admin-2fa` (<https://github.com/swe-himelrana/django-admin-2fa>) or `Django OTP` (<https://github.com/django-otp/django-otp>) to implement 2FA in our admin panel. These packages will require users to provide a one-time password (OTP) to access their accounts.

---

**Note** Let's try the Django Admin 2FA package. This package is using the `/admin/` URL by default (it's hardcoded in the package), so if you have set a custom admin URL, as we learned previously, you need to change it in the `settings.py` file OR you can download the package and change the URL in the package source code.

---



Let's install the package:

```
pip install django-admin-2fa
```

Add it to the `INSTALLED_APPS` in `settings.py`:

```
INSTALLED_APPS = [
 ...
 'djangoadmin2fa',
]
```

Add it to the `urls.py`:

```
path("admin2fa/", include("djangoadmin2fa.urls")),
```

Add the middleware in `settings.py`:

```
MIDDLEWARE = [
 ...
 'djangoadmin2fa.middleware.Admin2FAMiddleware',
 'djangoadmin2fa.middleware.Admin2faRateLimitMiddleware',
]
```

And run the migrations:

```
python manage.py migrate
```

Now, when we try to access the admin panel, we will be asked to set up 2FA using an authenticator app (Google Authenticator, Authy, etc.) and as shown in Figure 21-4 and Figure 21-5, to pass the verification process.

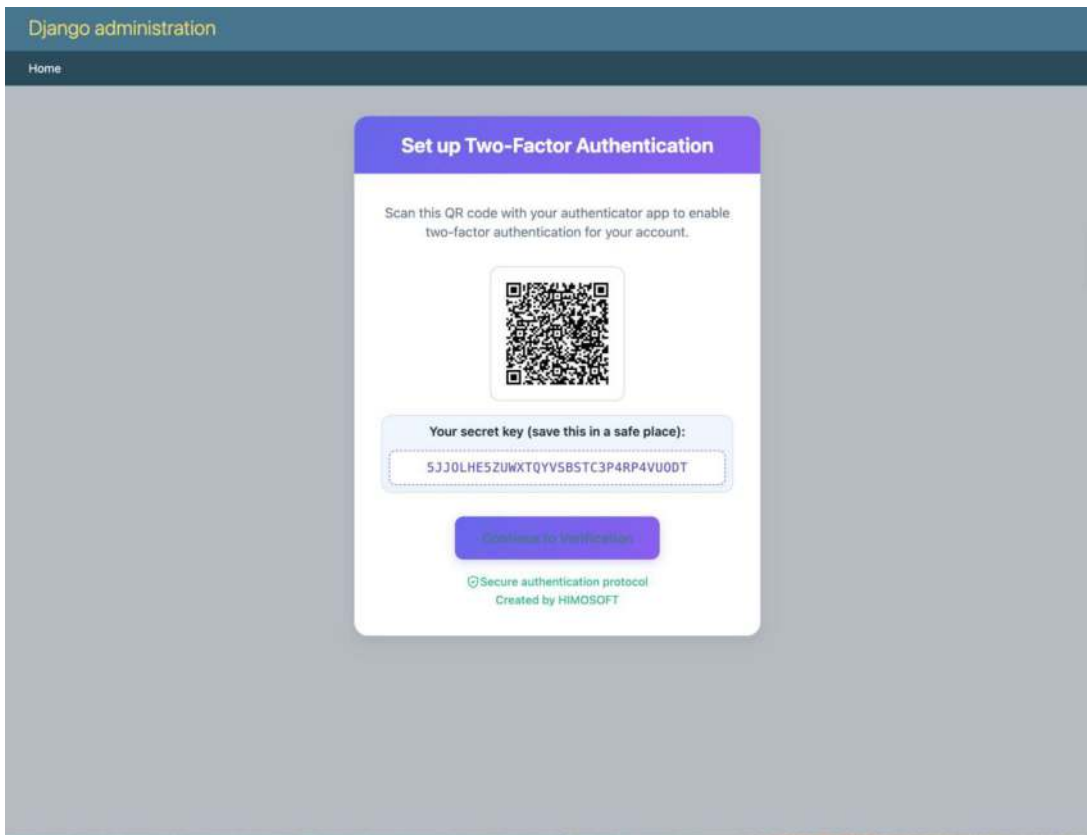


**Figure 21-3.** Admin 2FA setup screen (authenticator QR code/secret)

Then, if we try to access the admin panel again, we will be asked to enter a code.

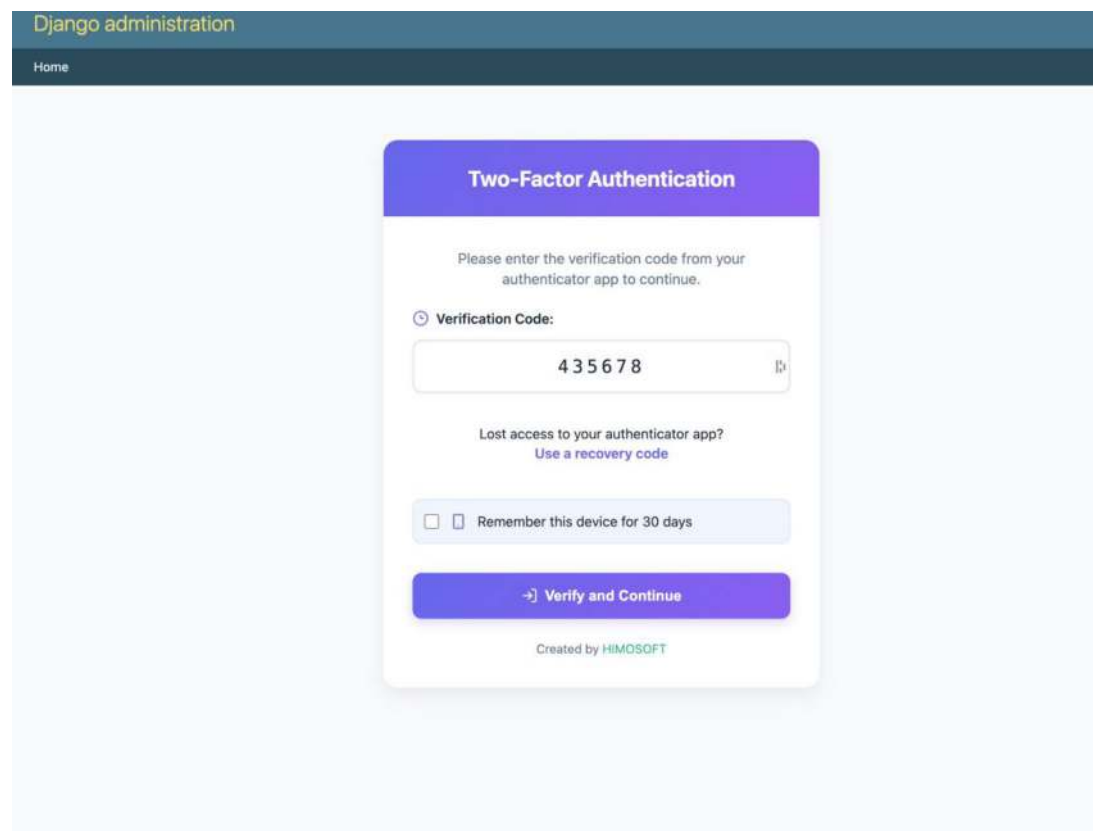


**Figure 21-4.** *OTP entry prompt on subsequent admin visits*

If we enter the wrong code, the access will be denied.

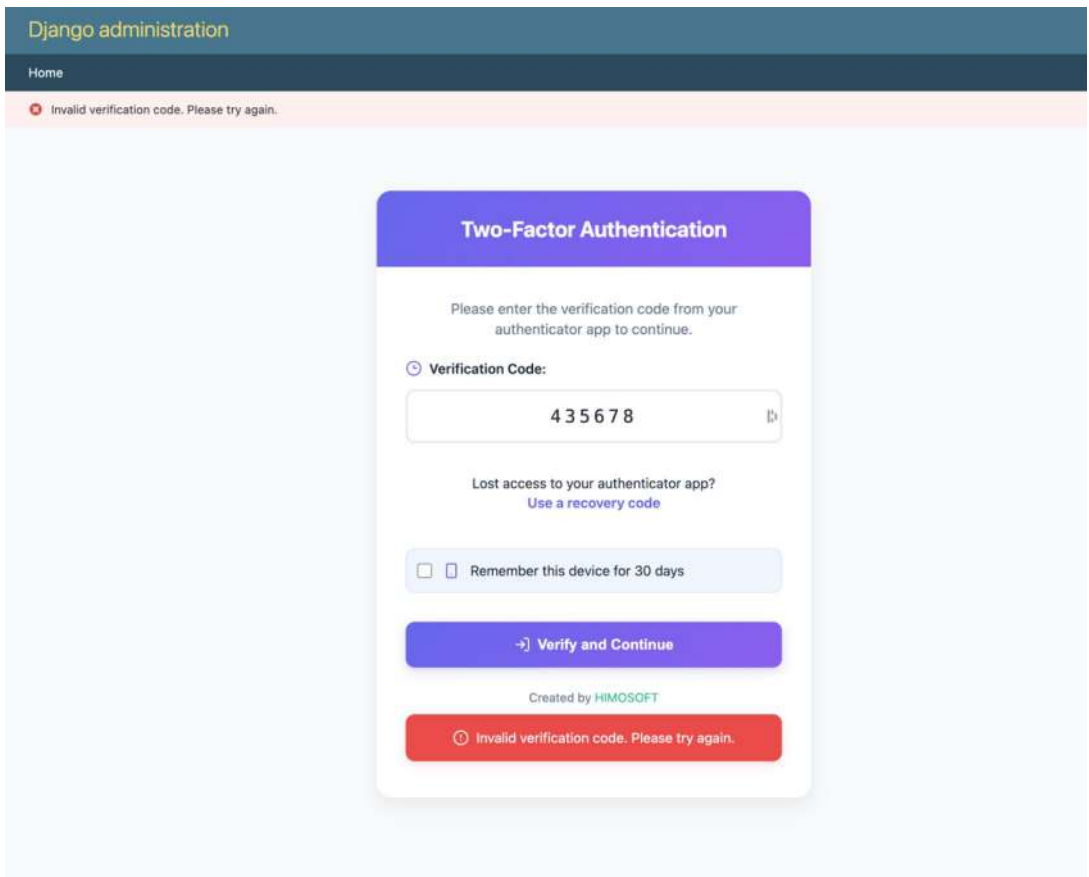


*Figure 21-5. Access denied after an invalid one-time password*

## Protect SECRET\_KEY

The Django SECRET\_KEY is one of the most critical security settings in your Django application (and often neglected). It's a 50-character random string that Django uses for cryptographic signing, session security, and various security-related operations.

## Why SECRET\_KEY Is Critical

The SECRET\_KEY is used by Django for

- **Cryptographic Signing:** Signing cookies and other values so Django can detect tampering (e.g., signed cookies and tokens).

- **Session Security:** Securing cookie-based sessions (they are signed, not encrypted). For database/cache-backed sessions, the cookie typically stores only a session ID.
- **CSRF Protection:** Generating and validating CSRF tokens to prevent cross-site request forgery attacks.
- **Password Reset Tokens:** Securing password reset (and similar) links via signed, time-limited tokens.
- **Key Derivation:** Deriving secret material used by some security features (signing, tokens, etc.).

If an attacker gains access to your `SECRET_KEY`, they can

- **Forge valid session cookies** and impersonate any user
- **Bypass CSRF protection** and execute unauthorized actions on behalf of users
- **Tamper with signed data** and make it appear authentic to Django
- **Generate valid tokens** for password resets and similar flows
- **Potentially access admin accounts** if other security measures are weak

## Common `SECRET_KEY` Exposure Risks

### 1. Version Control Systems

Never commit your `SECRET_KEY` directly in your code:

*# X NEVER DO THIS*

```
SECRET_KEY = 'django-insecure-abc123def456ghi789jkl012mno345pqr678stu901vw
x234yz'
```

## 2. Public Repositories

If your code is public or shared, ensure the `SECRET_KEY` is not visible:

```
X AVOID - Still visible in git history
SECRET_KEY = os.getenv('SECRET_KEY', 'fallback-key-for-development')
```

## 3. Environment Files

Don't commit `.env` files containing secrets:

```
X Never commit this file
echo "SECRET_KEY=your-actual-secret-key" > .env
git add .env # DON'T DO THIS
```

# Best Practices for SECRET\_KEY Management

## 1. Use Environment Variables

Store your `SECRET_KEY` in environment variables:

```
settings.py
import os

In production, do not generate a fallback key at runtime:
the value must be stable across restarts.
SECRET_KEY = os.environ.get("SECRET_KEY")
if not SECRET_KEY and not DEBUG:
 raise ValueError("SECRET_KEY environment variable must be set in
production")
```

## 2. Generate Strong Keys

Use Django's built-in key generator:

```
Generate a new secret key
python manage.py shell
>>> from django.core.management.utils import get_random_secret_key
>>> print(get_random_secret_key())
```

### 3. Separate Development and Production

Use different keys for different environments:

```
settings.py
if DEBUG:
 # Convenient for local dev; use an environment variable if you want it
 # to persist across restarts.
 SECRET_KEY = os.environ.get("SECRET_KEY", "dev-secret-key-for-
 development-only")
else:
 SECRET_KEY = os.environ['SECRET_KEY']
```

## Emergency Response

If your SECRET\_KEY is compromised:

1. **Immediately generate a new key**
2. **Force reauthentication** (log everyone out)
3. **Review access logs** for suspicious activity
4. **Monitor for unauthorized access** attempts
5. **Consider the scope of the breach** and notify affected users if necessary

---

Rotating SECRET\_KEY invalidates signed data (password reset tokens, signed cookies, CSRF tokens, etc.). It does **not** necessarily invalidate database/cache-backed sessions, because those typically rely on a random session ID stored in a cookie. To log everyone out, clear server-side sessions (e.g., by deleting session records) and restart your services so the new key is in effect.

---

## Example Implementation

Here's a secure setup for your settings.py:

```
import os
from django.core.management.utils import get_random_secret_key

SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = os.environ.get('SECRET_KEY')
if not SECRET_KEY:
 if DEBUG:
 SECRET_KEY = get_random_secret_key()
 else:
 raise ValueError("SECRET_KEY environment variable must be set in
production")

Additional security settings
```

---

**Remember** Your SECRET\_KEY is like the master key to your application's security. Protect it as if your entire application depends on it—because it does!

---



## CHAPTER 22

# Permission Systems

In Django, permissions are the rules that decide which users may perform which actions in your project. They apply everywhere your code checks access, and they matter most in the Django Admin, where staff manage data through the built-in interface.

A clear permission setup keeps sensitive information safe and limits each user to the views and actions their role allows.

## Permission Models

Django's built-in authentication system is based on two core concepts:

- Users: Individual accounts
- Groups: Collections of users sharing the same permissions

Each model in Django automatically comes with four default permissions:

- `add_<modelname>`: Create new objects.
- `change_<modelname>`: Update existing objects.
- `delete_<modelname>`: Remove objects.
- `view_<modelname>`: View objects.

You can view and edit these in the **Admin > Users** or **Admin > Groups** sections. Let's have a look at the default permissions for the **User** model (see Figure 22-1 and Figure 22-2).

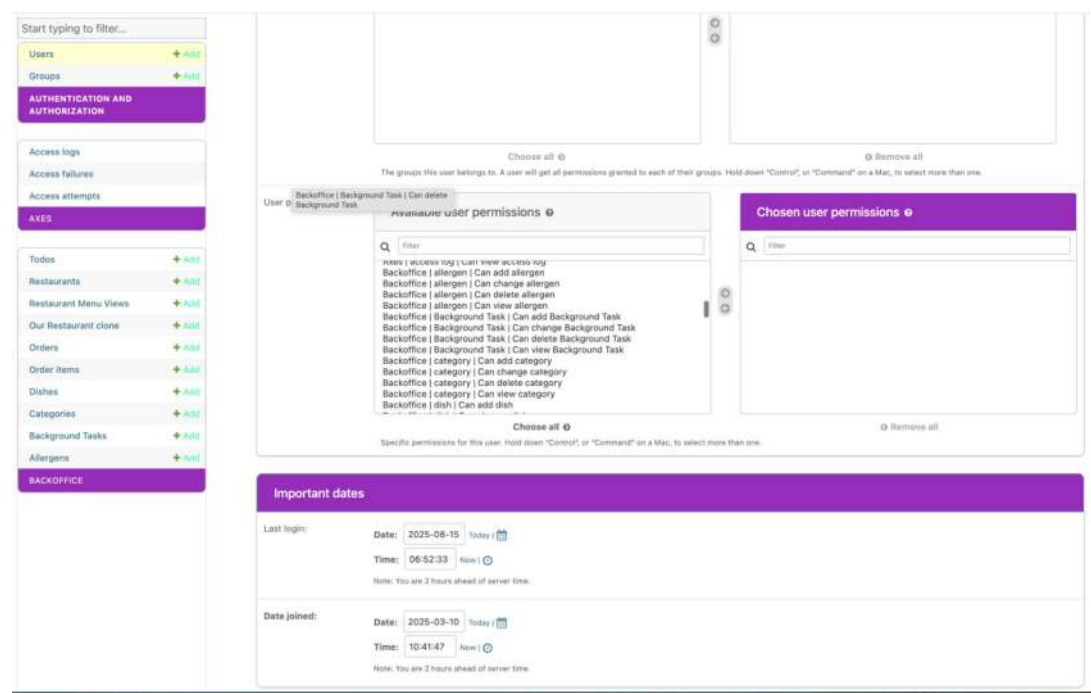


**Figure 22-1.** Default User model permissions in the admin (superuser: none listed individually)

When editing a user, we can see the default permissions for the User model.

As my user is a superuser, I will have automatically all permissions for all models, which is why no permissions are displayed in the “Chosen user permissions” section.

This superuser configuration can be seen in my user details page, on the “Permissions” tab.

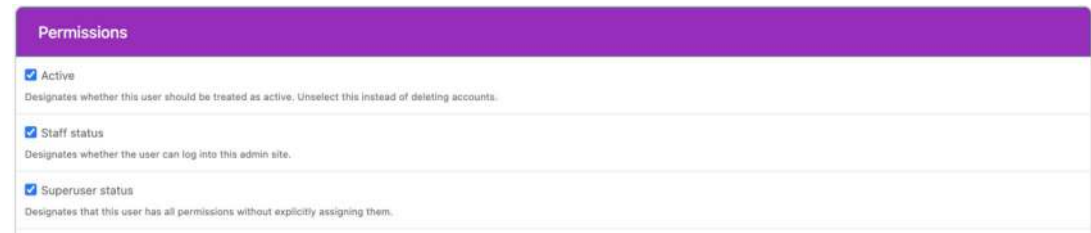


**Figure 22-2.** User details Permissions tab showing staff and superuser flags

Remember we created our superuser with the following command:

```
python manage.py createsuperuser
```

which set both fields **is\_staff** and **is\_superuser** to True.

**is\_staff** is a boolean field that indicates if the user is a staff member and can access the admin panel. **is\_superuser** is a boolean field that indicates if the user is a superuser and has all permissions.

## Creating a New User

To see how permissions play out in real use, let's create a new user in the Admin (see Figure 22-3 and Figure 22-4), then grant him permissions.

The screenshot displays the Django Admin interface for adding a new user. The sidebar on the left contains navigation links: 'Users', 'Groups', 'AUTHENTICATION AND AUTHORIZATION', 'Access logs', 'Access failures', 'Access attempts', 'AXES', 'Todos', 'Restaurants', 'Restaurant Menu Views', 'Our Restaurant clone', 'Orders', 'Order items', 'Dishes', 'Categories', 'Background Tasks', 'Allergens', and 'BACKOFFICE'. The main content area is titled 'Add user' and includes the following elements:

- Username:** A text input field with a required field indicator. Below it, a note states: 'Required: 150 characters or fewer. Letters, digits and @/./+/-/\_ only.'
- Password-based authentication:** A section with two radio buttons: 'Enabled' (selected) and 'Disabled'. Below the buttons, a note explains: 'Whether the user will be able to authenticate using a password or not. If disabled, they may still be able to authenticate using other backends, such as Single Sign-On or LDAP.'
- Password:** A text input field with a required field indicator. Below it, four password requirements are listed:
  - Your password can't be too similar to your other personal information.
  - Your password must contain at least 8 characters.
  - Your password can't be a commonly used password.
  - Your password can't be entirely numeric.
- Password confirmation:** A text input field with a required field indicator. Below it, a note states: 'Enter the same password as before, for verification.'
- Buttons:** At the bottom, there are three buttons: 'SAVE' (purple), 'Save and add another' (blue), and 'Save and continue editing' (blue).

**Figure 22-3.** Add user form in Django admin

We will create a new user called **userdemo** and set his password. Once done, the user is created, and we are redirected to the user details page.

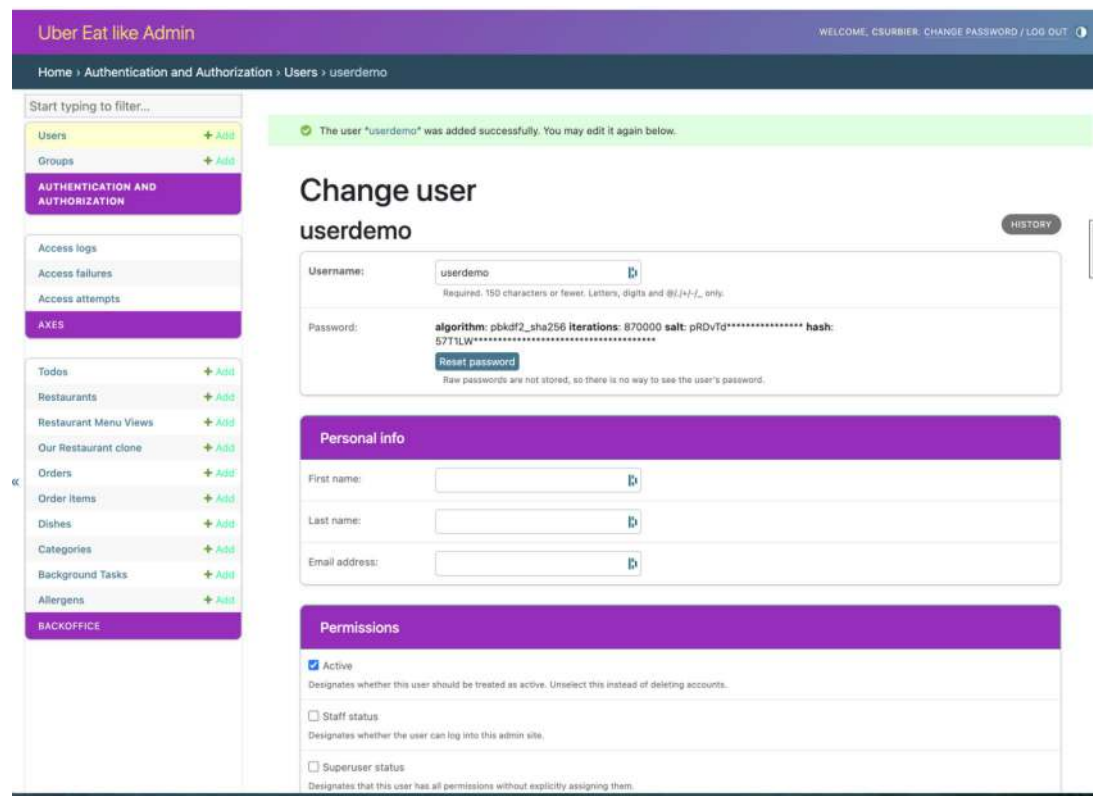


**Figure 22-4.** New user record after creation (user details page)

We will set the **is\_staff** field to True to allow the user to access the admin panel; **however**, we will not set the **is\_superuser** field to True to prevent the user from having all permissions, and we will not assign any permissions to the user as shown in Figure 22-5.

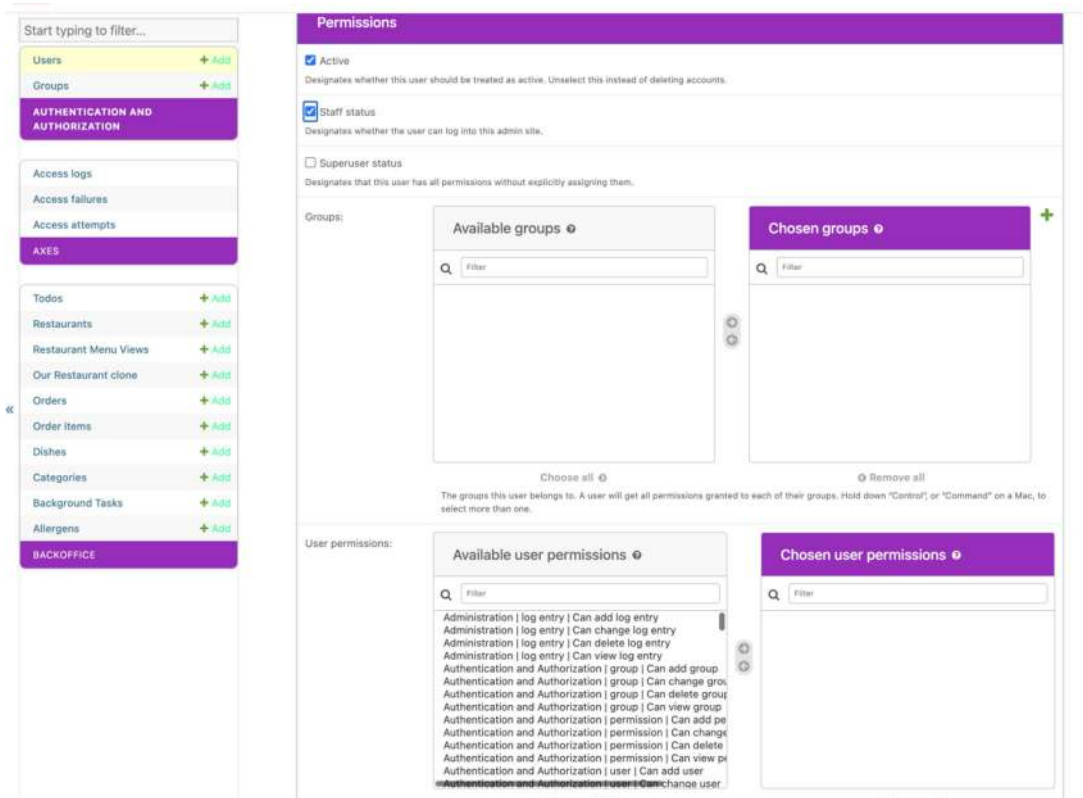


**Figure 22-5.** Staff user without superuser and with no explicit permissions yet

Now let's try to access the admin panel (see Figure 22-6) with the new user (you can access from another browser to keep your current session opened).

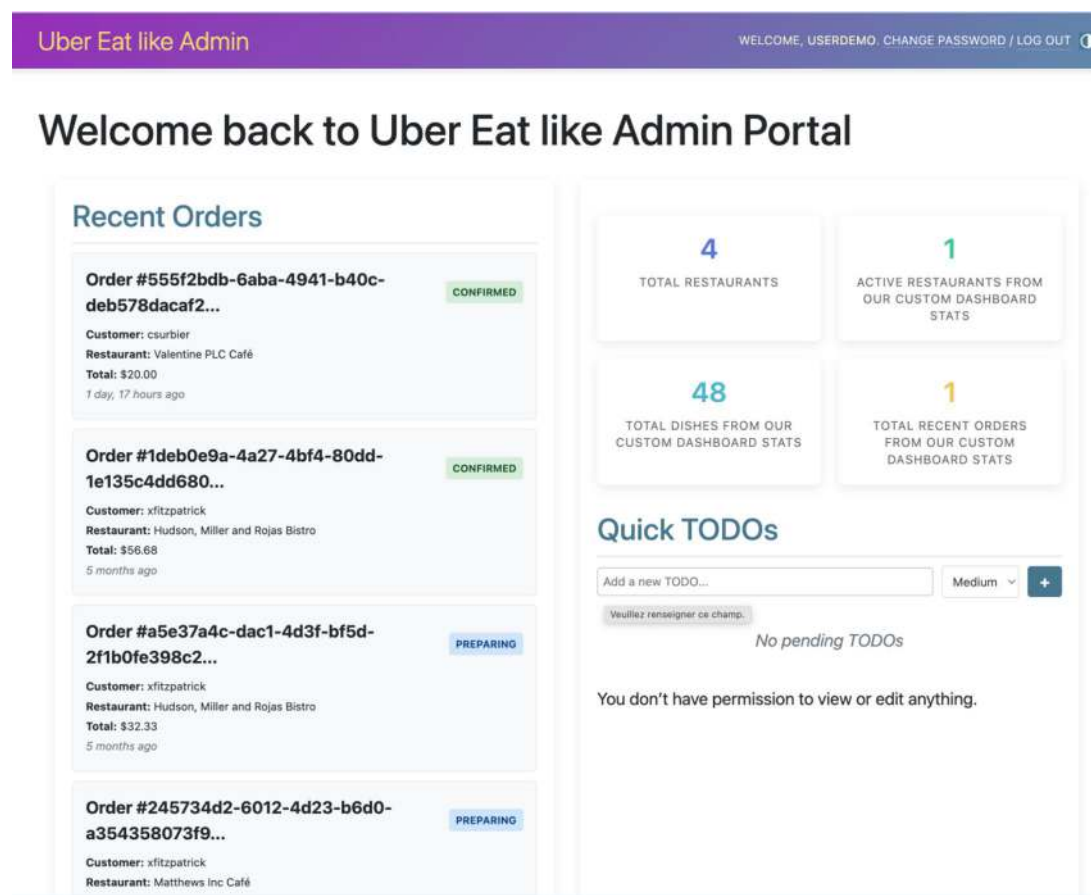


**Figure 22-6.** Limited admin index for a staff user with no model permissions

The **userdemo** can see our “Recent orders” or “Statistics” dashboard or “Quicks’s TODO” because we do not set any permissions restrictions on these components (we will learn more later on how to do this), but it is not able to see our application models (like **Order** or **Category** or **Restaurant**) and have a “You don’t have permission to view or edit anything.” message.

Let’s add some view permissions to the user (Figure 22-7).

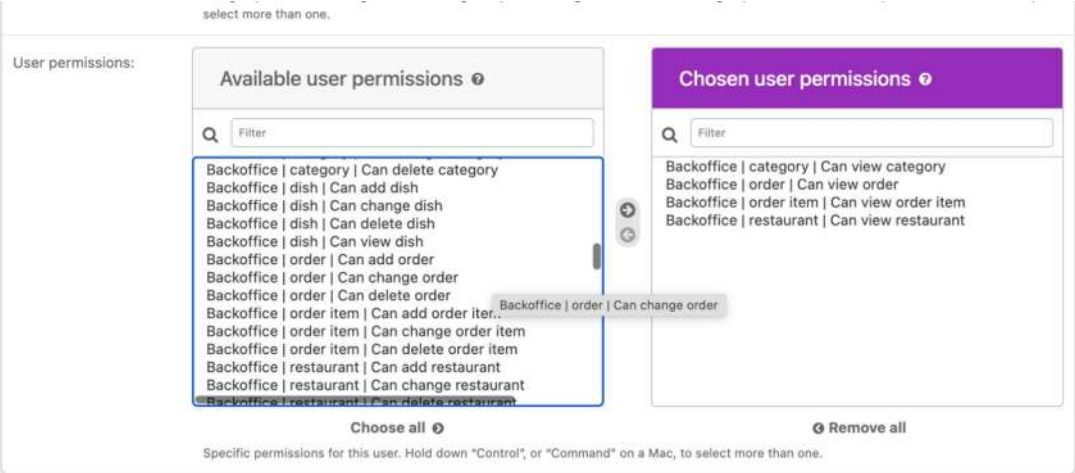


**Figure 22-7.** Granting view permissions to the user for backoffice models

If we reload our admin page of the **userdemo** user, we will see that he can now see the added models but only in the “View” mode (as shown in Figure 22-8).

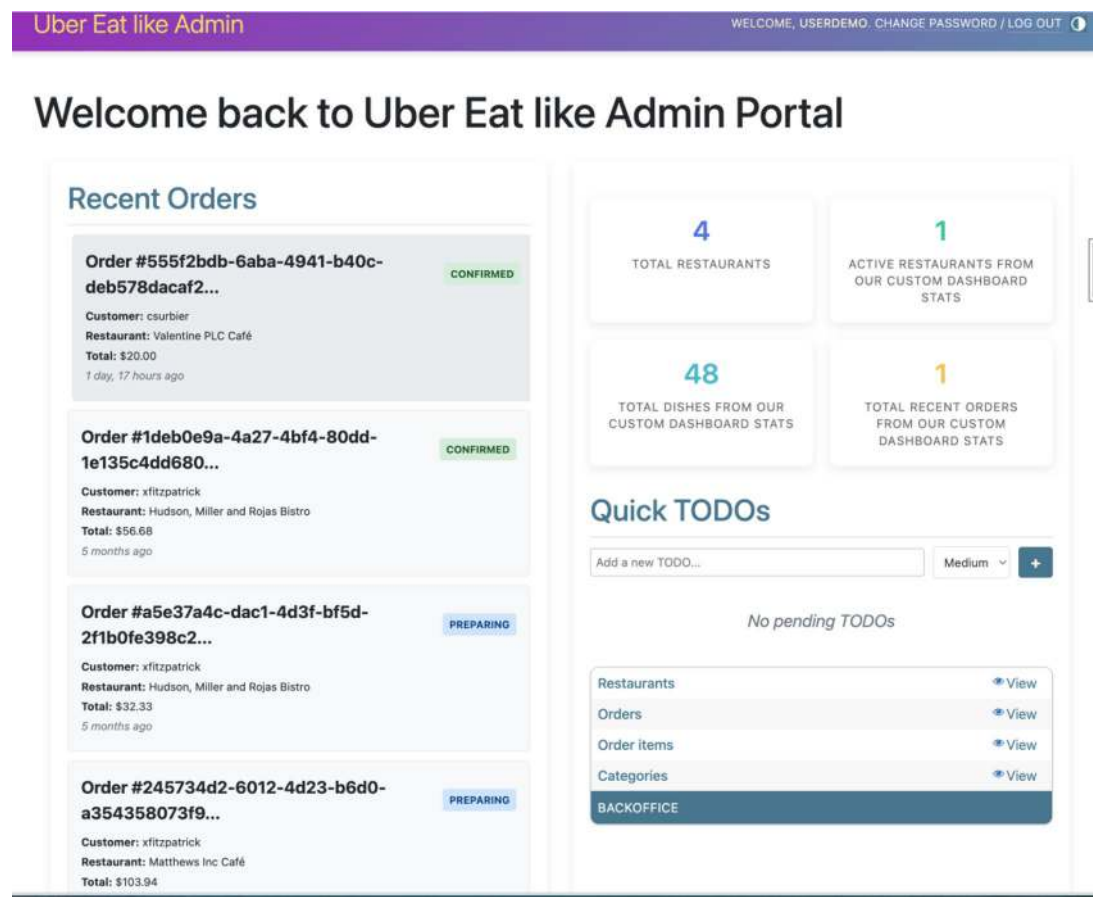


Figure 22-8. Backoffice models visible in view-only mode after permission grants

Having walked through that example, you should now have a clear picture of how permissions shape what each user can see and do in the Django Admin. We just need to add more permissions (such as add, change, delete) to the user to give him more access to the admin panel.

## Group Permissions

Specifying permissions for each user can be time-consuming and error-prone. A better approach is to use **groups**.

A group is a collection of users who will share the same permissions such as **SUPERADMIN** or **VIEWER** or **MARKETING\_DEPARTMENT...**



We can add permissions to a group and then assign that group to one or more users.

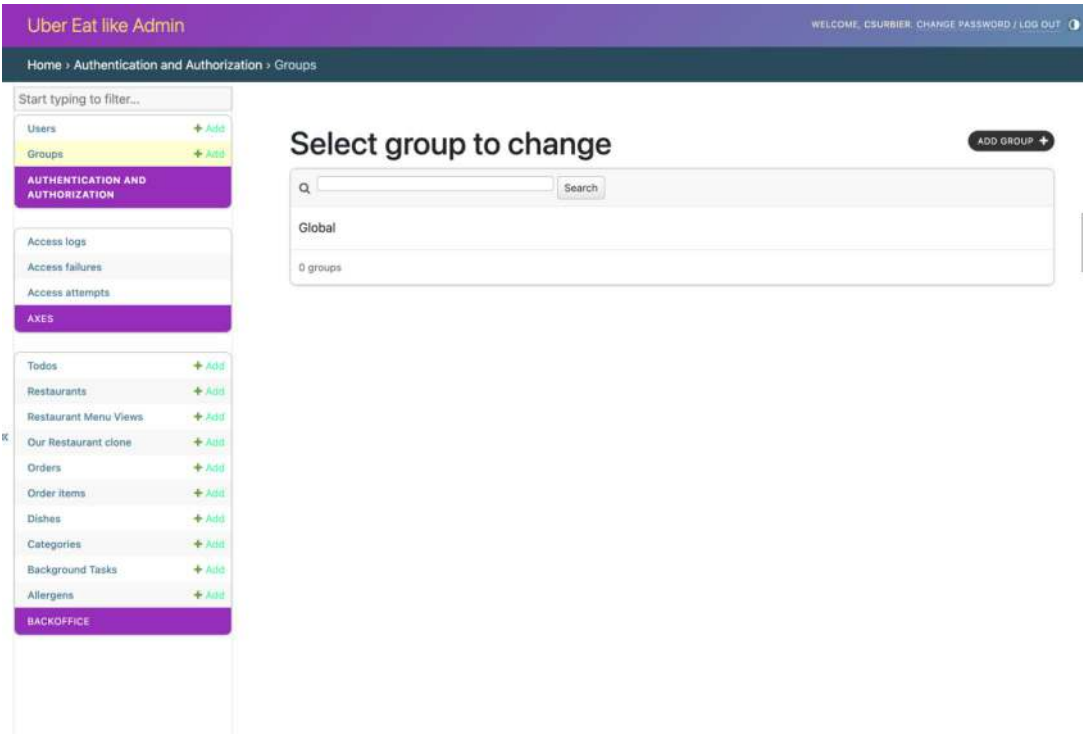


**Figure 22-9.** Add group form in Django Admin

We will create a new group called **viewers** and add all the backoffice models view permissions to it (please check Figure 22-9 to 22-12 for complete workflow).

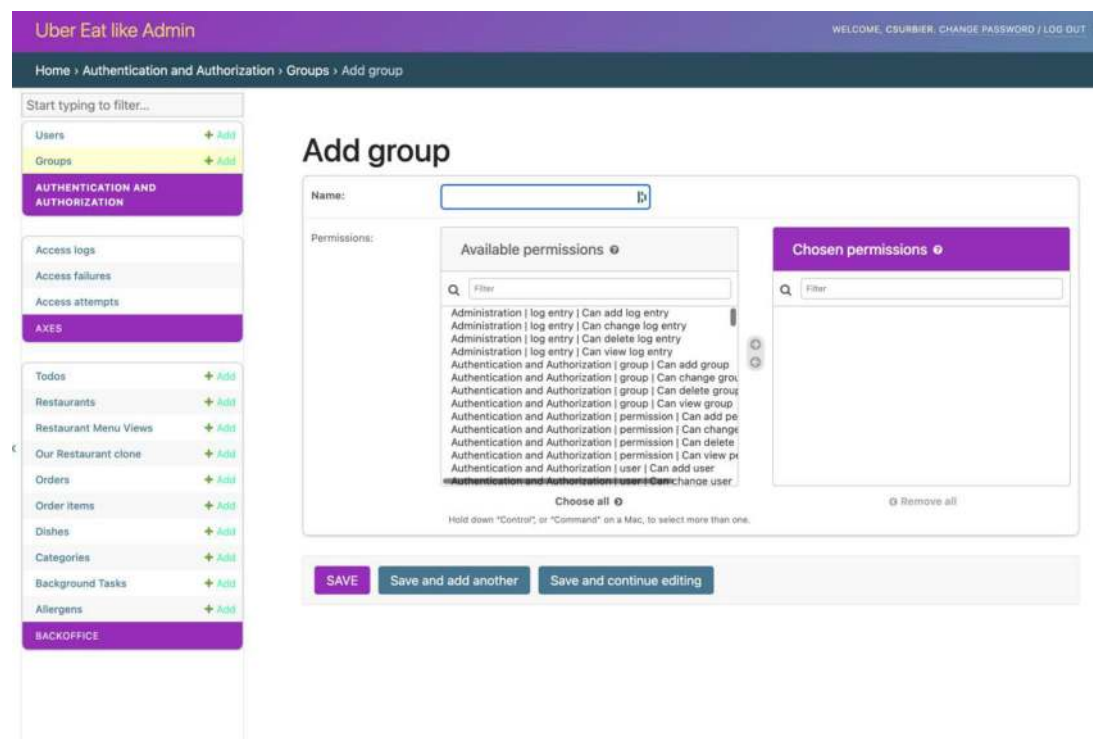


**Figure 22-10.** *Creating the viewers group and selecting model permissions*

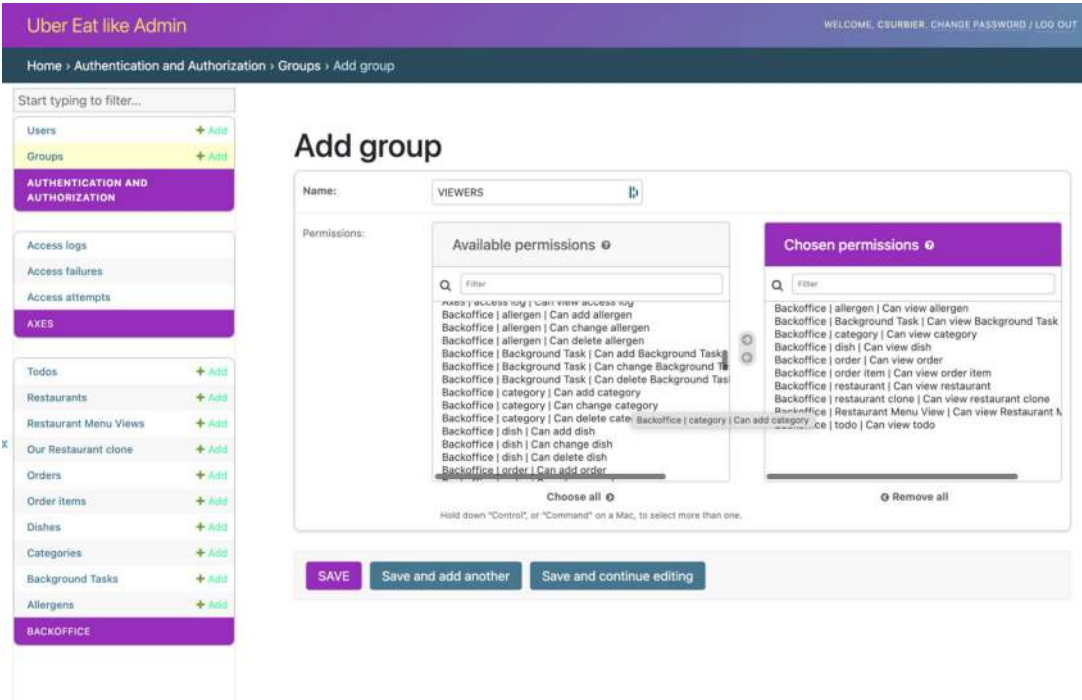


**Figure 22-11.** Group permissions: backoffice models with view access

and assign it to the **userdemo** user.

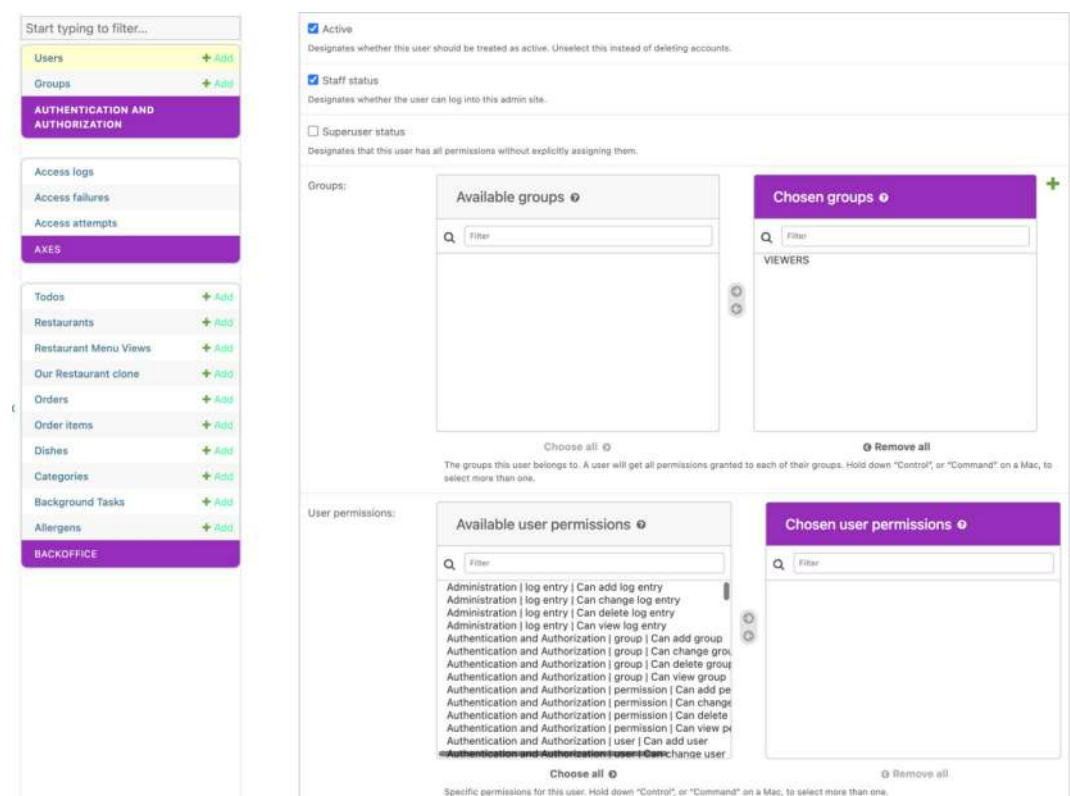


**Figure 22-12.** Assigning the viewers group to the userdemo account

Now if we reload the admin page of the **userdemo** user, we will see (as shown in Figure 22-13) that our user now has all the backoffice models in “View” mode.

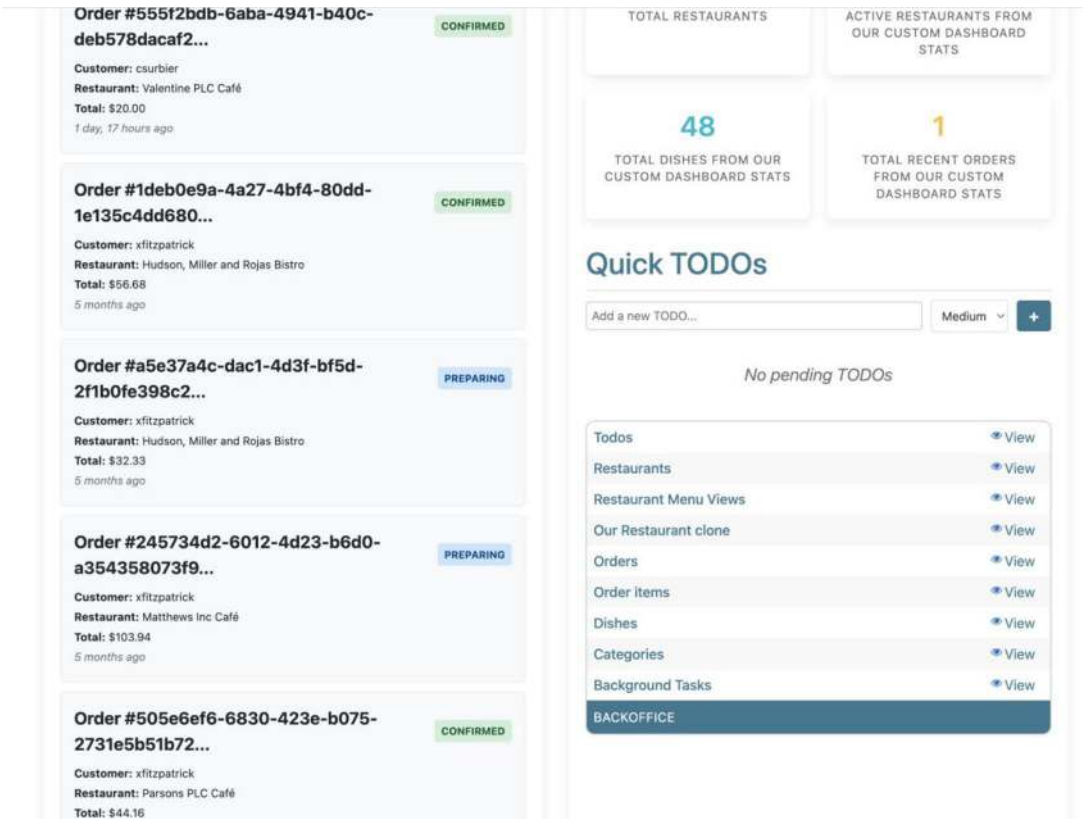


**Figure 22-13.** Admin index for userdemo after inheriting group-based view permissions

## CHAPTER 23

# Role-Based Access Control

In Django Admin, access control means restricting or granting visibility and actions for certain models, fields, or records:

- We can control which models appear in the Admin for each user.
- We can control what actions they can take (add, change, delete, view).
- Fine-grained control can also hide or disable certain fields for some users.

Let's have a look at the **Todo** model admin:

```
@admin.register(Todo)
class TodoAdmin(admin.ModelAdmin):
 list_display = ('title', 'description', 'is_completed')
 search_fields = ('title', 'description')
 list_filter = ('is_completed',)
 list_per_page = 10
 list_max_show_all = 200
 show_full_result_count = False
 date_hierarchy = 'created_at'

 def has_view_permission(self, request, obj=None):
 return request.user.is_superuser
```

With the code above, Django will deny **view** access to the **Todo** model for the **userdemo** user (see Figure 23-1), because `has_view_permission()` returns `False` (the user is not a superuser).

Note that this only overrides the **view** permission. If the user still has **add**, **change**, or **delete** permissions for **Todo** (e.g., via a group), the model may still appear in the admin and they may still be able to perform those actions. If your goal is to fully hide/lock down

the model, override the other permission hooks too (or override `get_model_perms()` to return an empty dict for unauthorized users).

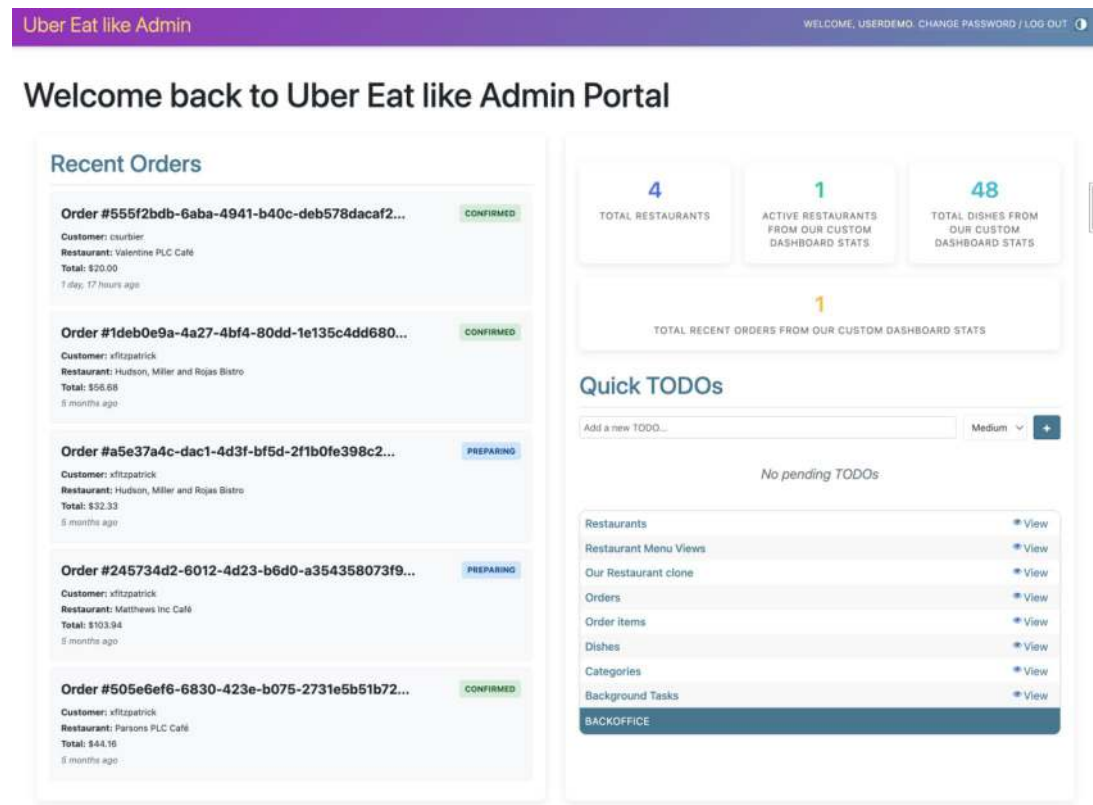


**Figure 23-1.** *Todo model hidden from non-superusers via `has_view_permission`*

If the user previously bookmarked the Todo changelist URL, they will now get a **403 Forbidden** (see Figure 23-2) when trying to access it without the required permission.

---

## 403 Forbidden

**Figure 23-2.** *HTTP 403 Forbidden when accessing Todo changelist without permission*

---

Moreover, even if the user is in a group such as **viewers**, your `ModelAdmin` override takes precedence for the **view** check. If you override only `has_view_permission()`, be explicit about what you intend to allow for add/change/delete as well.

---

Available methods to override permissions are

- `has_view_permission`
- `has_add_permission`
- `has_change_permission`
- `has_delete_permission`
- `has_permission`
- `has_object_permission`



**To avoid future issues, I will recommend to use** group permissions and avoid overriding permissions at model level.

---

## Row-Level Access (`get_queryset`)

We can also limit which records each user can see by overriding the `get_queryset` method.

Let's imagine we want to limit the records to let users see only the restaurants that they own (as shown in Figure 23-3). In our `RestaurantAdmin` class, we can override the `get_queryset` method and apply a filter to the queryset to only return the restaurants that the user owns:

```
def get_queryset(self, request):
 qs = super().get_queryset(request)
 if not request.user.is_superuser:
 qs = qs.filter(owner=request.user)
 return qs
```

---

With that in place, the user will only see the restaurants that they own in the admin panel.

---

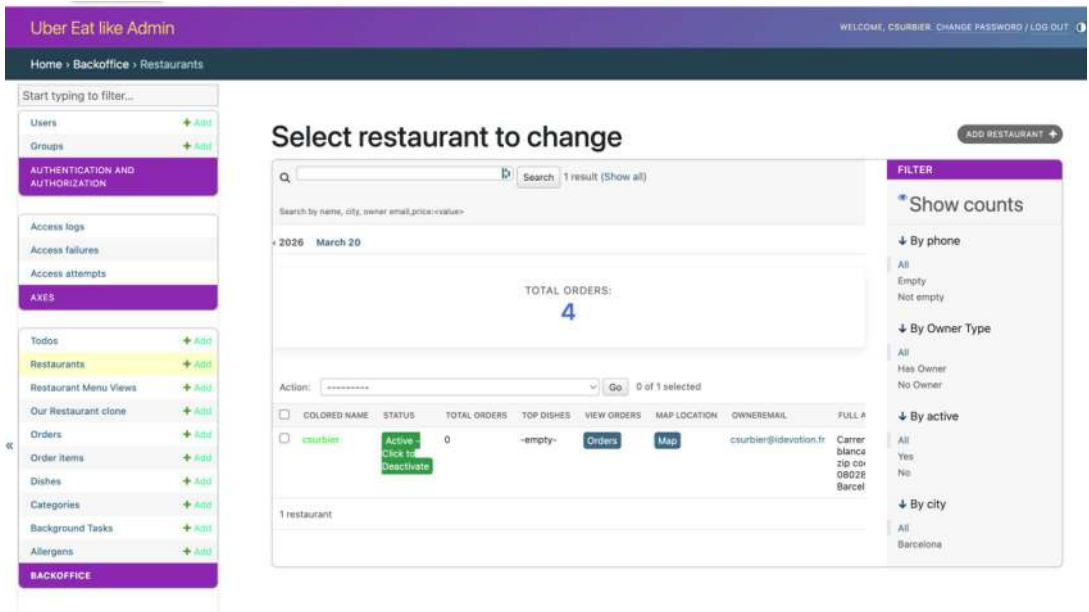


**Figure 23-3.** Restaurant changelist scoped to the current user via `get_queryset`

The user **csurbier** can only see his own restaurant in the admin panel.

Please note that because my user **csurbier** is a superuser, I had to remove the **is\_superuser** field from my user to be able to test the `get_queryset` method.

## Custom Permissions

Custom permissions are a way to define our own permissions for a model and define application-specific rights (beyond Django’s built-in add, change, and delete that we saw previously).

We declare them in the model’s Meta class via the `permissions` attribute; Django will then register each one in the admin with its codename (e.g., `orders.can_approve_order`).

For instance, if we want a user to be able to approve orders, we can add a `can_approve_order` permission to our `Order` model:

```
models.py
class Order(models.Model):
 ...

 class Meta:
 permissions = [
 ("can_approve_order", "Can approve order"),
]
```

When adding a new permission, we need to **run the migrations** to apply the changes to the database and be able to see the new permission in the admin panel:

```
python manage.py makemigrations
Migrations for 'backoffice':
 backoffice/migrations/0004_alter_order_options.py
 ~ Change Meta options on order
(venv) christophesurbier@csubier-2 DjangoAdmin % python manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, axes, backoffice, contenttypes,
 sessions, silk
Running migrations:
 Applying backoffice.0004_alter_order_options... OK
```

After that, as shown in Figure 23-4, we can grant the permission to a user or a group from the Django admin (go to **Users** or **Groups**, then add the permission to the user or the group).

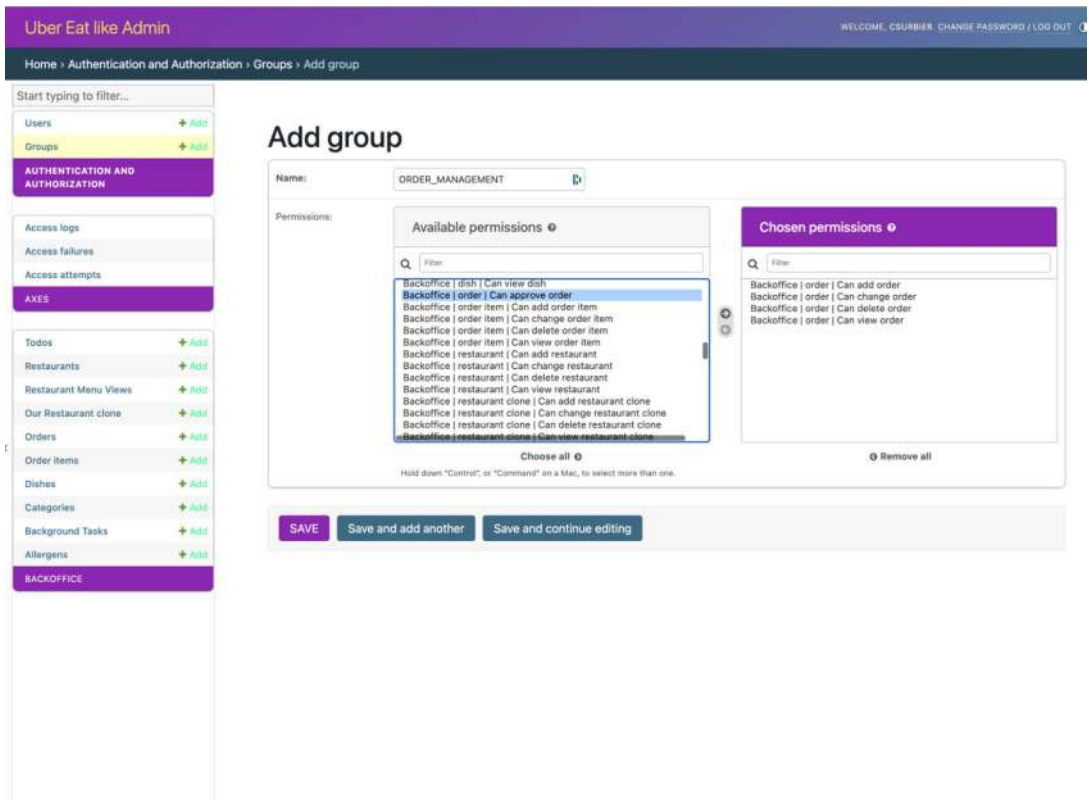


**Figure 23-4.** Custom model permission (`can_approve_order`) available for users or groups

We can now enforce the rule by checking the permission with `request.user.has_perm("orders.can_approve_order")` in our admin behavior. One common approach is to protect a custom admin action so only authorized users can run it; for example, an `approve_orders` action can raise **PermissionDenied** unless the user has the permission:

```
def approve_orders(self, request, queryset):
 if not request.user.has_perm("orders.can_approve_order"):
 raise PermissionDenied
 queryset.update(status="APPROVED")
```

---

The `approve_orders` action will only be available to users who have the `can_approve_order` permission.

---

## CHAPTER 24

# Audit Logging

**Django Admin** provides a built-in history log system that records all changes made to models in the admin interface. This is useful to track who made the changes, when they were made, and what was changed. You can access using the **History** button in the admin panel for a specific model.

Let's have a look at the history of the **Restaurant** model. First, I will modify my phone number as shown in Figure 24-1.

The screenshot shows the Django Admin interface for the 'Restaurant' model. The page title is 'Change restaurant csurbier from Barcelona'. The left sidebar shows a list of models under 'AUTHENTICATION AND AUTHORIZATION' and 'BACKOFFICE'. The main form has fields for Name, Image, Address, Zip code, City, Country, Phone, and Confirm phone. The 'Phone' field is highlighted with a blue border. Below the form is an 'Advanced options' section with a checkbox for 'Active' and a dropdown for 'Owner'.

**Figure 24-1.** *Editing a Restaurant field before opening admin history*

Then I can access the history (see Figure 24-2) from the model details page.

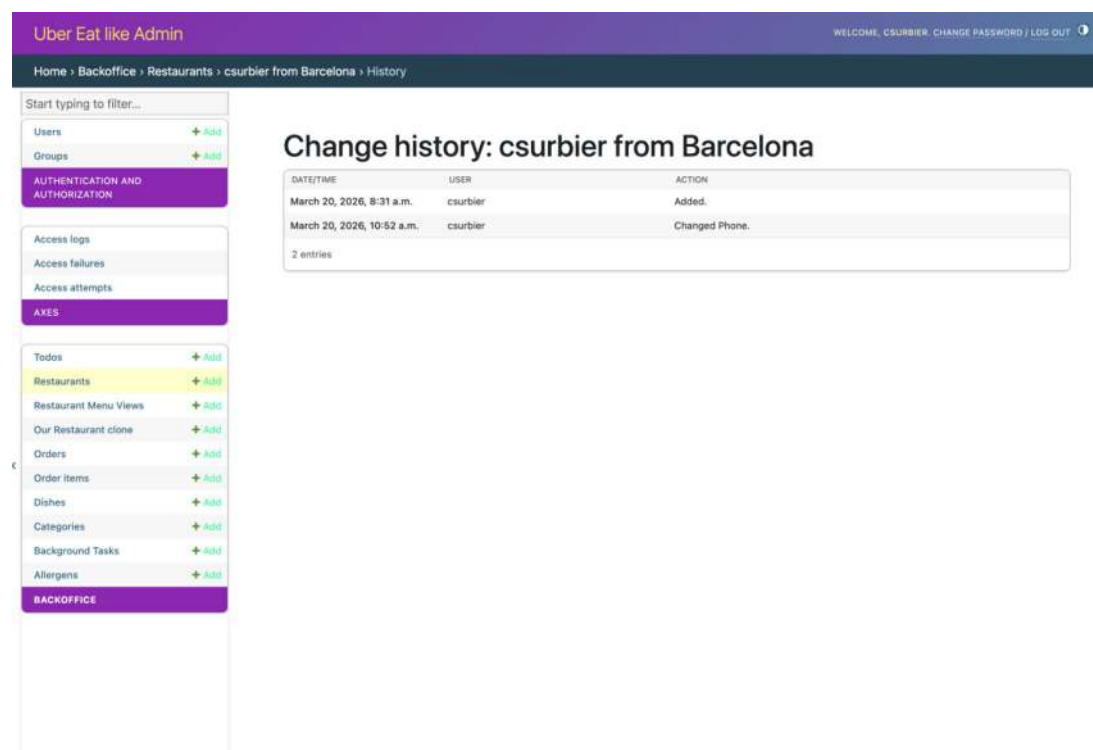


**Figure 24-2.** Django admin object history log for a Restaurant instance

The history will show you the changes made to the model.

# Audit Log

**Audit Logging:** `django-auditlog` (<https://django-auditlog.readthedocs.io/en/latest/installation.html>) is a Django package that goes a little bit further than the built-in history log system. It allows us to track changes made to models in the admin interface and more specifically to the objects of the model.

To install it, we can use the following command and/or add it to our requirements.txt file:

```
pip install django-auditlog
```

Then, we need to add it to our `INSTALLED_APPS` in our `settings.py` file:

```
INSTALLED_APPS = [
 ...
 'auditlog',
 ...
]
```

Then, we need to add it to our `MIDDLEWARE` in our `settings.py` file:

```
MIDDLEWARE = [
 ...
 'auditlog.middleware.AuditlogMiddleware',
 ...
]
```

Then, we need to run the migrations to apply the changes to the database and be able to see the new audit log in the admin panel:

```
python manage.py migrate
```

If we want to audit our **Restaurant** model, we need to add it to the auditlog registry. To do so, we can add the following code to our `models.py` file:

```
from auditlog.registry import auditlog

auditlog.register(Restaurant)
```

Then in our `admin.py` file, we need to add the following code:

```
@admin.register(Restaurant)
class RestaurantAdmin(AuditlogHistoryAdminMixin, admin.ModelAdmin):
 show_auditlog_history_link = True
```

which will add a new column **AuditLog** with a **View** button to the model `change_list` view as shown in Figure 24-3.

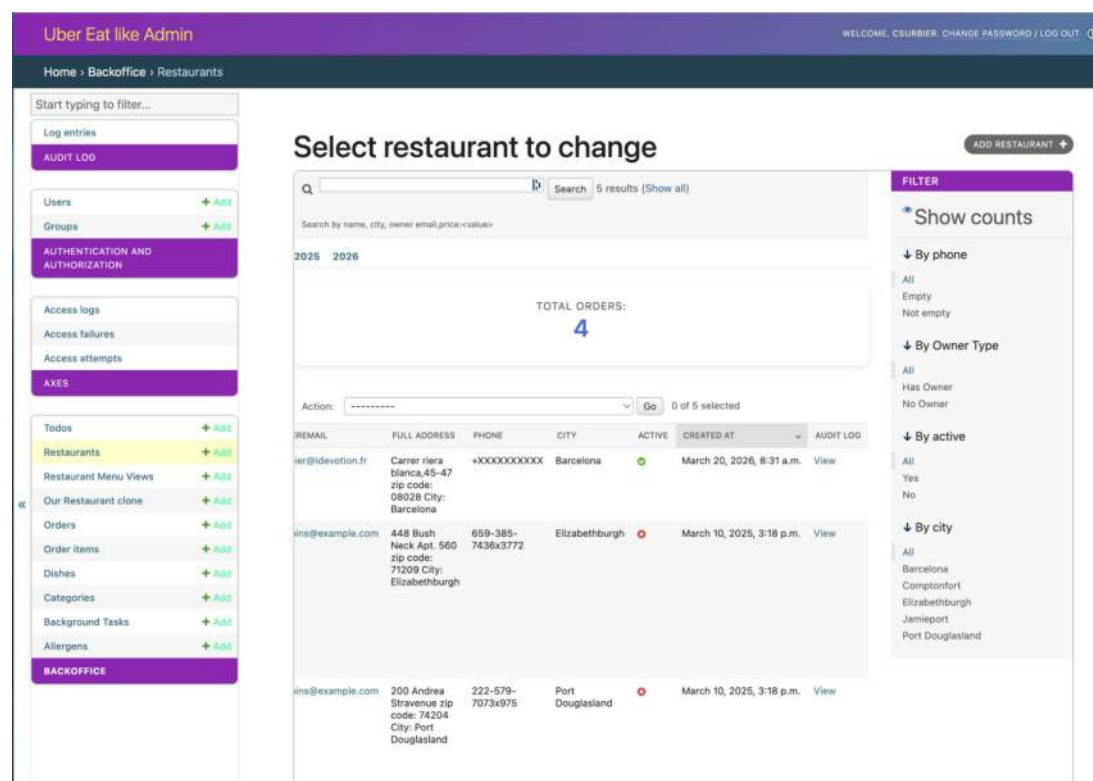


**Figure 24-3.** Django-auditlog “AuditLog” column with View link on Restaurant changelist

When we click the **View** button, we will see the audit log for the model (see Figure 24-4).



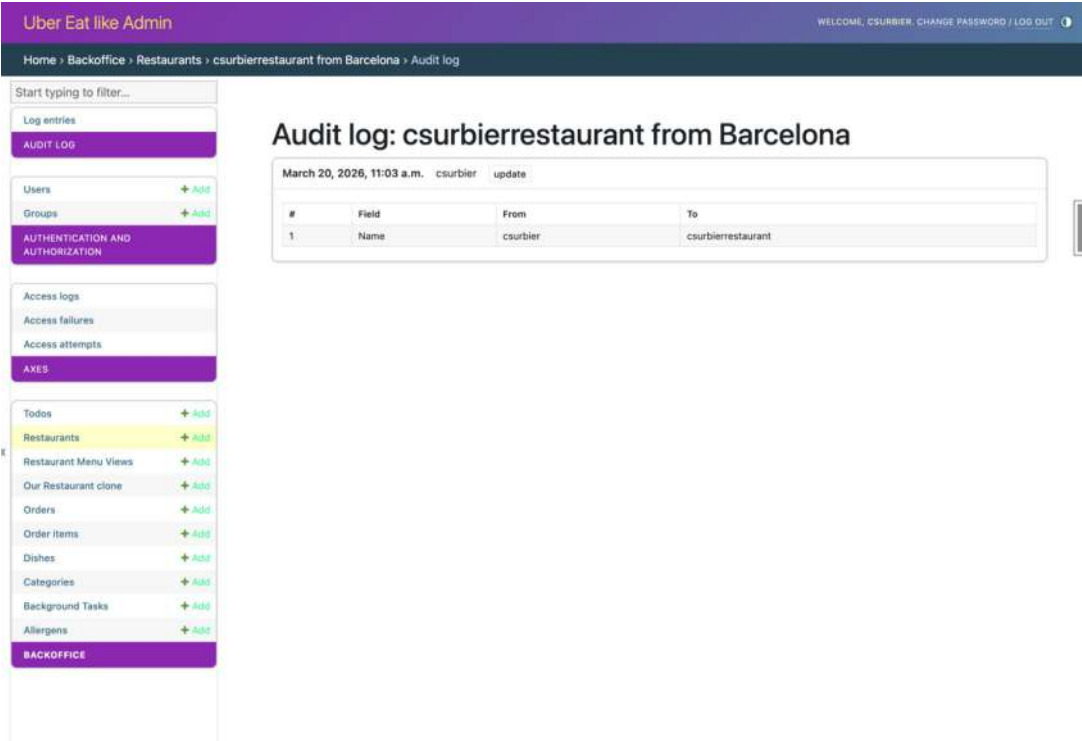


**Figure 24-4.** *Audit log entries for a restaurant (field-level change tracking)*

The audit log will show you the changes made to the model. As we can see in this example, I have changed the **name** of my restaurant to the value “**csurbierrestaurant**”.

Please refer to the documentation for more information on how to use the auditlog package.

## CHAPTER 25

# Security Best Practices

Django admin is built on standard Django views and templates, so it automatically benefits from many framework-level protections. However, the moment you customize `ModelAdmin` methods, list displays, or admin templates, you can accidentally reintroduce risk—especially with HTML rendering, custom forms, or raw SQL. The checklist below highlights what Django provides by default and what to keep an eye on in admin-heavy apps like ours.

## CSRF Protection

Django includes CSRF protection via `CsrfViewMiddleware` and the `{% csrf_token %}` template tag. By default, admin form submissions are protected because Django’s admin templates include the token. You only need to be careful when you introduce custom admin views/actions (e.g., intermediate POST steps, custom templates, or nonstandard endpoints): make sure the form includes `{% csrf_token %}`, and never disable CSRF middleware globally for “admin convenience.”

## XSS Prevention

Django templates escape variables by default, which prevents most reflected XSS issues when you render user-controlled content in admin pages. The main risk comes from outputting HTML from Python code. If you need to render HTML in `list_display/readonly_fields`, prefer `format_html()` so Django can safely mark only your intended markup as safe. Avoid using `mark_safe()` on values that can contain user input. When you must display rich text, ensure it is sanitized first (either at input time or with a trusted sanitization step) and keep admin templates with autoescaping enabled.

## SQL Injection

Django's ORM uses parameterized queries under the hood, which is the primary defense against SQL injection in typical admin workflows. In admin code, this usually means: build filters and queries using ORM lookups (`filter()`, `get()`, `annotate()`, etc.) and avoid constructing SQL strings manually. If you introduce raw SQL (or database-specific features), strictly use parameters/placeholders and never concatenate user input directly into the query.

## Security Headers

Security headers reduce the impact of certain classes of attacks by tightening browser behavior. For example, `SECURE_CONTENT_TYPE_NOSNIFF` helps prevent MIME-type sniffing, `X_FRAME_OPTIONS` reduces clickjacking by controlling whether the admin UI can be framed, and `SECURE_HSTS_*` enables HTTP Strict Transport Security so browsers prefer HTTPS. In admin contexts, these matter just as much as in the public site—especially if you embed admin pages in iframes (which is usually better avoided unless you have a strong reason and a safe configuration).

## Session Security

Admin actions typically have higher impact than normal user actions, so session hardening is worth doing. Django lets you control cookie behavior with settings like `SESSION_COOKIE_AGE` (e.g., keeping admin sessions shorter), `SESSION_SAVE_EVERY_REQUEST` (to extend session lifetimes with activity), `SESSION_COOKIE_SECURE` (send cookies only over HTTPS), `SESSION_COOKIE_HTTPONLY` (reduce access from JavaScript), and `SESSION_COOKIE_SAMESITE` (mitigate cross-site request risks). If you care about faster revocation, consider which `SESSION_ENGINE` you use (database-backed sessions are often easier to invalidate reliably than purely time-based cookies or cache-only strategies).

## Password Policy

Django ships with a default set of password validators configured via `AUTH_PASSWORD_VALIDATORS`. Those validators already cover several practical requirements (length, common passwords, numeric-only passwords, etc.). For admin accounts, treat “strong password” as a hard requirement: keep the validators enabled, and optionally add a custom validator for your chosen policy (e.g., length and complexity). The goal is simple: raise the cost of credential guessing and reduce account takeover risk.

## Production Security Checklist

We already covered most of these points in the previous chapters. As a reminder, here are the main points:

- Disable debugging and lock down host and transport settings: set `DEBUG = False`, configure `ALLOWED_HOSTS`, enable `SECURE_SSL_REDIRECT`, and if you are behind a reverse proxy load balancer, set `SECURE_PROXY_SSL_HEADER` correctly.
- For long-lived HTTPS enforcement, configure `SECURE_HSTS_*` thoughtfully.
- Treat secrets as runtime configuration: keep `SECRET_KEY` (and any “admin URL” or other sensitive routing values) in environment variables rather than in source control.

Before going live, ensure your Django security posture is production-ready.

## Summary

In this part, we focused on securing the Django Admin by reducing its attack surface and tightening access controls.

- We hardened access to the admin entry point (custom URLs, IP/network controls, and brute-force protection on the login).
- We added stronger authentication and safer transport defaults (2FA and HTTPS enforcement with Django’s security settings and your edge proxy/load balancer).

- We reviewed core secret management practices (`SECRET_KEY` handling, rotation impact, and incident response steps).
- We applied least-privilege principles in the admin (group permissions, careful use of `ModelAdmin` permission hooks, and row-level scoping with `get_queryset()`).
- We closed with a security checklist covering CSRF, XSS, SQL injection, headers, sessions, and password policy—so your admin stays secure as you customize it.

# Conclusion

When we started this book, we logged into a plain Django Admin interface showing two links—**Users** and **Groups**—and not much else. By the time you reached this page, that same framework had become a full-featured business operations platform: a branded, responsive backoffice capable of managing restaurants, menus, and orders in real time, protected by a layered security system, instrumented with charts and PDF reports.

That transformation did not require a single line of frontend framework code, no React, no Vue, no custom API server built from scratch. It required a deep understanding of what Django Admin already gives us and the knowledge of how to extend it purposefully.

## What We Built Together

In **Part I, “Foundations,”** we set the project up from scratch and learned how Django Admin actually works under the hood: how models are registered, how the AdminSite orchestrates everything, and how template inheritance and URL routing tie it all together. We designed our Uber Eats-like data model—Restaurant, Category, Dish, Order, OrderItem—which became the backbone of every example throughout the book.

In **Part II, “Essential Features,”** we went from a raw list of database rows to a genuinely usable interface. We customized list displays, added search and filters, built inline editing for related models, wrote form validation rules, and tackled performance from the start with `select_related`, `prefetch_related`, database indexes, and caching strategies. A well-configured admin with poor query performance is not a production-ready admin.

In **Part III, “Advanced Customization,”** we pushed beyond what most developers ever explore. We built custom admin URLs and intermediate action pages, integrated a REST API directly inside the admin, offloaded heavy operations to Celery background tasks, wrote raw SQL optimizations for complex reporting queries, and used database views and model proxies as powerful tools for presenting data in nonobvious ways.

## CONCLUSION

In **Part IV, “UI/UX Enhancement,”** we transformed the visual identity of the interface. We override core admin templates block by block, integrated Bootstrap and Tailwind, added HTMX for seamless real-time interactions, built a custom dashboard with a TODO widget and live order feed, and implemented Chart.js visualizations alongside professional PDF report generation with ReportLab.

In **Part V, “Security and Access Control,”** we hardened the entire interface for production. We moved the admin URL, added IP filtering and rate limiting with `django-axes`, enabled two-factor authentication, secured the `SECRET_KEY` with environment variables, implemented group-based permissions and row-level access control, added audit logging with `django-auditlog`, and reviewed the full CSRF, XSS, SQL injection, and session security landscape.

## Django Admin or a Custom Backoffice?

I raised this question in the very first pages of the book, and now you have everything you need to answer it for your own project.

Django Admin is the right choice when your primary users are **internal staff** who need to manage data reliably and efficiently, when your backoffice requirements are largely CRUD-shaped with some business logic layered on top, and when development speed and maintainability matter more than pixel-perfect custom design. With the techniques from this book, you can take that interface remarkably far.

A custom backoffice becomes the better choice when the interface is customer-facing or public, when the user experience needs to diverge significantly from what a model-centric interface can offer, or when the volume of customization required to bend Django Admin to your needs exceeds the cost of building something purpose-built. At that point, Django Admin’s strengths—its conventions, its automatic form generation, its built-in auth—become constraints.

The honest truth is that most projects benefit from **both**—a custom public-facing product frontend and a Django Admin backoffice for internal operations. Django Admin handles the operational complexity so you can focus your custom development budget where it actually creates competitive advantage.

## Keep Exploring

The Django ecosystem around the admin keeps evolving. Packages like **django-unfold**, **django-jazzmin**, and **django-daisy** continue to improve and add new capabilities. The HTMX and Alpine.js ecosystems are expanding the range of interactions you can add without writing a single line of JavaScript framework code. Django itself is shipping performance and security improvements with every release.

The best next step is to take a real project—yours, or one you are building—and apply what you have learned chapter by chapter. Nothing accelerates understanding faster than solving a real problem.

Thank you for reading. I hope this book saves you weeks of trial and error and helps you build backoffice interfaces you are genuinely proud of.

—**Christophe Surbier**



# Index

## A

Actions responses, [76–78](#)  
Adaptive layouts  
    description, [323](#)  
    media queries, [323–326](#)  
Admin index, [338–339](#)  
Admin index page changes, [183–188](#)  
Admin interface, [26–29](#), [161–167](#)  
admin.py file, [27–29](#), [108–114](#), [177](#), [330](#)  
Admin site, [180–182](#)  
AdminSite Class, [9](#)  
Admin templates, [268–271](#)  
Admin URL, [369–371](#)  
Ajax integration, [209–211](#)  
Ajax operations, [308–315](#)  
Ajax request, [195–208](#)  
api/restaurant-stats endpoint, [229–232](#)  
Asset management, [9](#)  
@register decorator, [18–20](#)  
Audit Logging, [408–411](#)  
Auth\_user model, [97](#)

## B

Backend engineers, [4](#)  
Background tasks  
    Celery, [237–245](#)  
    task management, [246–249](#)  
Backoffice, [294](#)  
Backoffice admin templates, [342](#)  
Backoffice application, [8](#)  
base.html template, [259](#)

BaseInlineFormSet class, [155](#)  
base\_site.html, [308](#)  
Block override patterns  
    block tags work, [263–264](#)  
    index.html, [261–263](#)  
    inheritance chain, [259](#)  
    template base.html, [259–263](#)  
    template resolution order, [264–265](#)  
Block tags, [265](#)  
Bootstrap cdn version, [285–287](#)  
Bootstrap integration  
    Bootstrap locally, [287–288](#)  
    Tailwind CSS, [288–293](#)  
Brute-force protection, [373–374](#)  
Built-in actions, [73–74](#)  
Built-in authentication system, [385](#)

## C

Cascading filters, [125](#)  
Category models, [83](#)  
CategorySerializer, [233](#)  
cdn version, [285](#)  
Celery integration, [237–245](#)  
Chain filters, [125–130](#), [136–140](#)  
Change form, [62](#)  
change\_form.html template, [169](#)  
Changelist context, [348](#)  
Change list template, [342–345](#)  
changelist\_view method, [189–195](#)  
change\_view attributes  
    foreign key restriction, [151–152](#)  
    forms widget, [149–150](#)

## INDEX

- change\_view attributes (*cont.*)
  - inline formset, [152–156](#)
  - inline pagination, [158–160](#)
  - prefilled values, [146–149](#)
  - raw\_id\_fields, [156–158](#)
  - restrictions, [160](#)
- Chart.js integration, [342–345](#)
- Chart payload, [347–348](#)
- Charts and analytics, [341–342](#)
  - Chart.js integration, [342–345](#)
  - order chart data, [346–347](#)
- Class-based approach, [17](#)
- Classes attribute, [71](#)
- Column label, [117](#)
- Configuration
  - model class, [25](#)
- CRUD operations, [3](#)
- CSRF protection, [413](#)
- CSS Grid, [323](#)
- CSV export, [349–354](#)
- cursor.execute() method, [252](#)
- cursor.fetchall() method, [252](#)
- Custom actions, [74–76](#)
- Custom dashboards, [327–329](#)
  - endpoints and views creation, [331–333](#)
  - TODO model, [329–331](#)
  - TODO widget template, [333–338](#)
- Custom filters, [55–60](#), [272–275](#)
- custom\_index method, [266](#)
- Customization, [9](#)
- Custom list\_display, [121–125](#)
- Custom methods, [47](#)
- Custom permissions, [403–405](#)
- Custom public-facing, [418](#)
- Custom template, [163](#)
- Custom template tags, [265](#)
  - admin templates, [268–271](#)

- creation, [266–268](#)
  - practices, [271–272](#)
- Custom themes, [294–305](#)
- Custom validation, [152–156](#)

## D

- Database indexes, [93–103](#)
- Database tables, [211](#)
- Data migration, [129](#)
- Debugging, [253–255](#)
- Default admin index page
  - changes, [183–188](#)
- Default values, [146–149](#)
- Deployment, [365](#)
- Designing models, [10–17](#)
- DishAdmin class, [209–211](#)
- Dish list, [136](#)
- DishSerializer, [233](#)
- Display
  - change list, [60–62](#)
  - custom filters, [55–60](#)
  - customizing field names, [39–43](#)
  - date, [38–39](#)
  - list\_editable, [51–52](#)
  - list options, [37–38](#)
  - list\_per\_page, [46–47](#)
  - related field filters, [54–55](#)
  - related model, [43–46](#)
  - search fields, [62](#)
- Django Admin, [3](#), [28](#), [73](#), [79](#), [146](#), [158](#), [285](#),  
[298](#), [323](#), [399](#), [407](#), [413](#), [418](#)
  - interface, [17](#)
  - REST API (*see* REST API integration)
  - template files, [170–172](#)
  - URL management, [226](#), [369–371](#)
- django.contrib.staticfiles, [364](#)

Django Daisy, [280–283](#), [419](#)  
 Django debug toolbar, [253–255](#)  
 Django developers, [4](#)  
 Django DRF framework, [236](#)  
 Django Grapelli, [277](#)  
 Django Jazzmin, [277–278](#), [419](#)  
 Django Jet, [276–277](#)  
 Django staticfiles, [364](#)  
 Django-Tailwind package, [289](#)  
 Django toolbar, [97](#), [102](#)  
 Django-Unfold, [419](#)

## E

Environment files, [381](#)  
 Environment variables, [381](#)  
 Exclude attribute, [64–66](#)

## F

Fake data, [20–24](#), [132–136](#)  
 Fake models, [211–226](#)  
 fetch('/api/order-charts/') approach, [347](#)  
 Field-level validation, [82](#)  
 Fields attribute, [62–63](#)  
 Fieldsets, [66–71](#)  
 Filters, [52–54](#)  
 Fixed-length unique value, [130](#)  
 Flexbox, [323](#)  
 Foreign key restriction, [151–152](#)  
 Form customization, [79–81](#)  
 Form widgets, [149–150](#)  
 Full-stack developers, [4](#)

## G

get\_queryset, [402–403](#)  
 get\_search\_results() method, [142](#)  
 get\_urls method, [209](#)

get\_urls() method, [122](#)  
 Group permissions, [392–397](#)

## H

has\_view\_permission(), [399](#)  
 HTMX, [308–315](#)  
 HTMX component, [319–322](#)  
 HTTP 403 Forbidden, [400](#)  
 HTTPS and SSL/TLS encryption, [375](#)

## I

index.html file, [261–263](#), [309](#), [319](#), [338–339](#)  
 Inline editing  
     categories, [84](#), [85](#)  
     many to many relationship, [86–92](#)  
 Inline formset, [152–156](#)  
 InlineModelAdmin class, [86](#)  
 Inline pagination, [158–160](#)  
 Installation, [5–8](#)  
 Integration, [9](#)  
 Intermediate confirmation page, [161–167](#)  
 Internationalization, [10](#), [29–32](#)  
 IP blocking/filtering, [371–372](#)

## J, K

JavaScript integration, [307](#)  
     Ajax operations, [308–315](#)  
     HTMX operations, [308–315](#)  
 JSON payload, [347–348](#)  
 json\_script filter, [347–349](#)

## L

list\_display  
     chain filter, [125–130](#)  
     custom admin URLs and  
         actions, [121–125](#)

## INDEX

`list_display` (*cont.*)

    custom link, [117–120](#)

    database view, [140–141](#)

    proxy model, [141–142](#)

`list_display_links`, [49–50](#)

`list_editable`, [51–52](#)

`list_filter`, [99](#)

Login form changes, [172–179](#)

lookups method, [139](#)

## M

Many to many field, [90](#)

Many to many

    relationship, [86–92](#)

Material design, [293](#)

Media queries, [323–326](#)

`menu_grid.html` template, [216](#)

`message_user` method, [76–78](#)

Messages, [76–78](#)

Meta class, [140](#)

Metadata leakage, [129](#)

Meta options, [25](#)

Mobile-first approach, [323](#)

`ModelAdmin`, [80](#)

`ModelAdmin` class, [9](#), [17–20](#), [52](#)

`ModelAdmin` method, [117](#)

`model_admin` object, [58](#)

`ModelAdmin` options, [108](#)

Model-centric interface, [418](#)

Models, [10](#)

Model-specific settings, [25](#)

`mytheme.css` file, [303](#)

## N

N+1 queries problem, [93](#)

## O

One-time password (OTP), [375](#)

Optional dependency, [355](#)

Order chart data, [346–347](#)

Ordering instruction, [47–49](#)

`ownlogin.html` file, [178–179](#)

## P

Pagination handling, [106–107](#)

Parsing, JSON, [347–348](#)

Password policy, [415](#)

PDF reports

    imports and optional dependency, [355](#)

    pie chart, [355–363](#)

Performance optimization

    caching strategies, [103–106](#)

    pagination, [106–107](#)

    query, [93–103](#)

Permission models, [385–387](#)

    group permissions, [392–397](#)

    new user creation, [387–392](#)

    superuser configuration, [386](#)

Pie chart, [355–363](#)

PostgreSQL, [131–132](#)

`prefetch_related`, [102–103](#)

Profiling tools, [253–255](#)

Public repositories, [381](#)

Python dict, [347–348](#)

`python-dotenv`, [131](#)

## Q

Query optimization

    debugging, [253–255](#)

`prefetch_related`, [102–103](#)

raw SQL optimization, [251–253](#)  
 select\_related, [93–102](#)

## R

raw\_id\_fields, [156–158](#)  
 Raw SQL optimization, [251–253](#)  
 Readonly\_fields, [63–64](#)  
 Redis, [237](#)  
 Reference lit, [364](#)  
 Registration, [10](#)  
 Related field filters, [54–55](#)  
 ReportingStats, [140](#)  
 Reportlab package, [354–355](#)  
 requirements.txt file, [237](#)  
 REST API integration  
     administrative functionality, [229–232](#)  
     serialization, [232–236](#)  
 RestaurantAdmin classes, [93](#), [103](#), [145](#),  
     [286](#), [292](#)  
 Restaurant model, [214](#), [407–409](#)  
 RestaurantSerializer, [233](#)  
 reverse() function, [122](#)  
 Row-level access (get\_queryset), [402–403](#)

## S

search\_fields attribute, [142–145](#)  
 search\_help\_text attribute, [145](#)  
 search\_term variable, [143](#)  
 SECRET\_KEY, [379–380](#)  
     emergency response, [382–383](#)  
     management, [381–382](#)  
     risks, [380](#)  
 Security, [10](#)  
 Security checklist, [415](#)  
 Security headers, [414](#)  
 SecurityMiddleware settings, [375](#)

select\_related, [93–102](#)  
 self.admin\_site.admin\_view() wrapper, [123](#)  
 Separate design topic, [129](#)  
 Serialization, [232–236](#)  
 Session security, [414](#)  
 settings.py, [253](#)  
 Simple caching strategies, [103–106](#)  
 SimpleListFilter class, [55](#)  
 SQL injection, [414](#)  
 SQL panel, [96](#)  
 Stacked format, [84](#)  
 StackedInline class, [83](#), [84](#)  
 Stacked single-column layout, [326](#)  
 STATIC\_URL, [365](#)

## T

TabularInline, [83](#)  
 Tailwind CSS, [288–293](#)  
 Task management system, [246–249](#)  
 tasks.py file, [240](#)  
 template file, [316–319](#)  
 Template inheritance, [9](#)  
     Ajax request, [195–208](#)  
     default admin index page  
         changes, [183–188](#)  
     default Django Admin files, [169–170](#)  
     default ordering, [180–182](#)  
     list view changes, [189–195](#)  
     login form, [172–179](#)  
 Template system  
     block override patterns, [259–265](#)  
     custom filters, [272–275](#)  
     custom tags, [265–272](#)  
     rebrand Django Admin, [276–283](#)  
 Todo changelist, [400](#)  
 TODO model, [329–331](#)  
 TODO widgets, [327](#), [333–339](#)

## INDEX

`toggle_active` method, [124](#)  
`toggle_active` view, [123](#)  
Two-factor authentication (2FA), [375–380](#)

## U

URL management, [226](#)  
    Ajax, [209](#)  
    faking models, [211–226](#)  
URL routing, [9](#)  
`urls.py` file, [315](#)  
`userdemo`, [388](#), [396](#)

## V, W

Validation rules, [81–82](#)  
Version control systems, [380](#)  
Viewers, [393](#)  
`view_orders` method, [286](#), [292](#)  
`view-restaurant-menu`, [219](#)  
`views.py` file, [315](#)

## X, Y, Z

XSS prevention, [413–414](#)