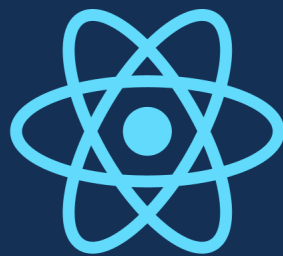




# EFFECTIVE REACT DEVELOPMENT WITH NX

Jack Hsu



# Effective React Development with Nx

A practical guide to full-stack React development in a monorepo

Jack Hsu and Juri Strumpflohner

This book is for sale at <http://leanpub.com/effective-react-with-nx>

This version was published on 2022-01-24



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 – 2022 Jack Hsu and Juri Strumpflohner

# Contents

<b>Introduction</b> . . . . .	<b>1</b>
Monorepos to the rescue! . . . . .	1
Why Nx? . . . . .	2
Is this book for you? . . . . .	2
How this book is laid out . . . . .	3
<b>Chapter 1: Getting started</b> . . . . .	<b>5</b>
Creating a Nx workspace . . . . .	5
Nx workspace configuration . . . . .	10
Nx commands . . . . .	15
Preparing for development . . . . .	18
<b>Chapter 2: Libraries</b> . . . . .	<b>21</b>
Apps and Libs . . . . .	21
The generate command . . . . .	24
Feature libraries . . . . .	24
UI libraries . . . . .	29
Using the UI library . . . . .	36
Data-access libraries . . . . .	38
Enforcing module boundaries . . . . .	45
<b>Chapter 3: Working effectively in a monorepo</b> . . . . .	<b>51</b>
The dependency graph . . . . .	51
Only recompute affected projects . . . . .	53
Computation Caching . . . . .	58
Adding the API application . . . . .	61
Automatic code formatting . . . . .	71
<b>Chapter 4: Bringing it all together</b> . . . . .	<b>72</b>

## CONTENTS

Checkout API and shared models . . . . .	72
Cart data-access library . . . . .	73
Cart feature library . . . . .	82
Wiring up add button in books feature . . . . .	86
Building for production . . . . .	92

# Introduction

If you've ever worked at a company with more than one team, chances are you've had to deal with some challenges when it comes to change management.

In a typical work setting, development teams are divided by domain or technology. For example, one team building the UI in React, and another one building the API in Express. These teams usually have their own code repositories, so changes to the software as a whole requires juggling multiple repositories.

A few problems that arise from a multi-repository (often called "Polyrepo") setup include:

- Lack of collaboration because sharing code is hard and expensive.
- Lack of consistency in linting, testing, and release processes.
- Lack of developer mobility between projects because access may be unavailable or the development experience varies too greatly.
- Difficulty in coordinating changes across repositories.
- Late discovery of bugs because they can only occur at the point of integration rather than when code is changed.

## Monorepos to the rescue!

A lot of successful organizations such as Google, Facebook, and Microsoft—and also large open source projects such as Babel, Jest, and React—are all using the monorepo approach to software development.

What is a monorepo though? A monorepo is a single repository containing multiple distinct projects, where we don't just have code collocation, but well-defined relationships among these projects. As you will see in this book, a

monorepo approach – when done correctly – can save developers from a great deal of headache and wasted time.

Still there are quite some misconceptions when it comes to monorepos.

- Monorepos are monolithic and not for building microservices and microfrontends<sup>1</sup>
- Continuous integration (CI) is slow
- “Everyone can change *my* code”
- Teams losing their autonomy

All of the above will be addressed throughout this book.

## Why Nx?

Nx is a fast, smart and extensible build system that helps teams develop applications at any scale. It integrates with modern frameworks and libraries, provides computation caching and smart rebuilds, as well as code generators.

## Is this book for you?

This book assumes that you have prior experience working with React, so it does not go over any of the basics. We will also make light use of the Hooks API, however understanding it is not necessary to grasp the concepts in this book.

Nx generates TypeScript code by default, so we’ll be using that in our examples throughout the book. Don’t fret if this is your first introduction to TypeScript. We will not be using any advanced TypeScript features so a good working knowledge of modern JavaScript is more than enough.

Consequently, this book might be for you if:

---

<sup>1</sup><https://blog.nrwl.io/misconceptions-about-monorepos-monorepo-monolith-df1250d4b03c>

- You just heard about Nx and want to know more about how it applies to React development.
- You use React at work and want to learn tools and concepts to help your team work more effectively.
- You want to use great tools that enable you to focus on product development rather than environment setup.
- You use a monorepo but have struggled with its setup. Or perhaps you want to use a monorepo but are unsure how to set it up.
- You are pragmatic person who learns best by following practical examples of an application development.

On the other hand, this book might not be for you if:

- You are already proficient at using Nx with React and this book may not teach you anything new.
- You *hate* monorepos so much that you cannot stand looking at them.

Okay, the last bullet point is a bit of a joke, but there are common concerns regarding monorepos in practice.

## How this book is laid out

This book is split into three parts.

In **chapter 1** we begin by setting up the monorepo workspace with Nx and create our first application—an online bookstore. We will explore a few Nx commands that work right out of the box.

In **chapter 2** we build new libraries to support a book listing feature.

In **chapter 3** we examine how Nx deals with code changes in the monorepo by arming us with intelligent tools to help us understand and verify changes. We will demonstrate these Nx tools by creating an `api` backend application.

In **chapter 4** we wrap up our application by implementing the `cart` feature, where users can add books to their cart for check out. We will also look at building and running our application in production mode.



# Chapter 1: Getting started

Let's start by going through the terminology that Nx uses.

## Workspace

A folder created using Nx that contains applications and libraries, as well as scaffolding to help with building, linting, and testing.

## Project

An application or library within the workspace.

## Application

A package that uses multiple libraries to form a runnable program. An application is usually either run in the browser or by Node.

## Library

A set of files that deal with related concerns. For example, a shared component library.

Now, let's create our workspace.

## Creating a Nx workspace

You can create the workspace as follows:

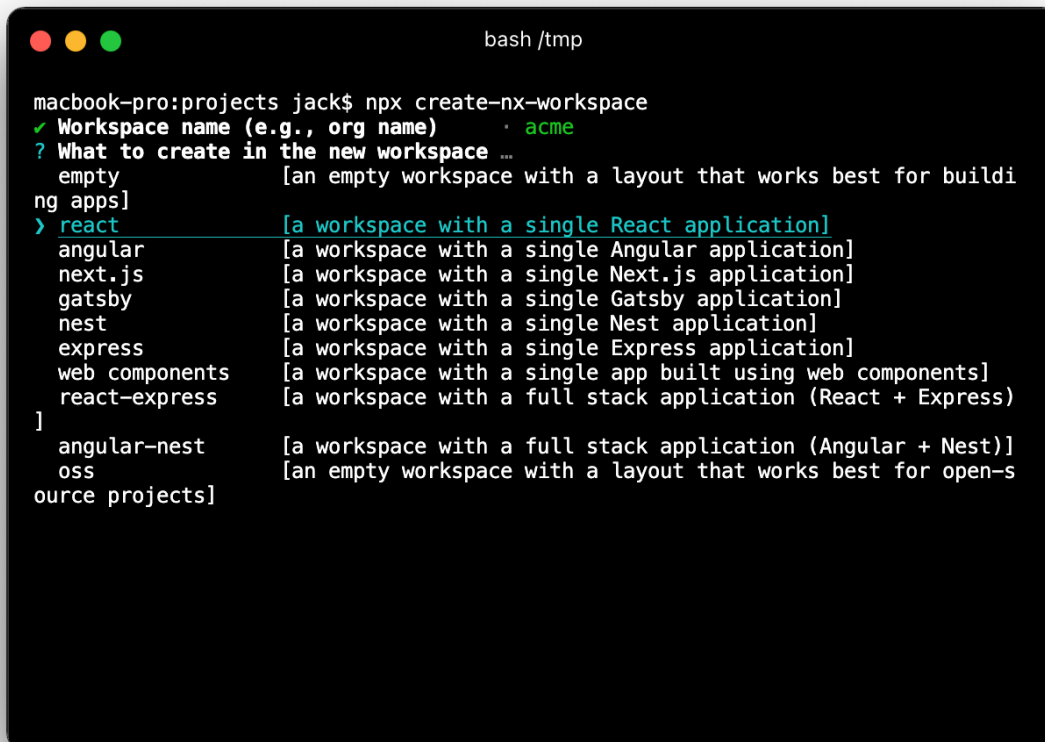
```
npx create-nx-workspace@latest
```



Note: The `npx` binary comes bundled with NodeJS. It allows you to conveniently install then run a Node binary without the need to install it globally.

Nx will ask you for a **workspace name**. Let's use `acme` as it is the name of our imaginary organization. The workspace name is used by Nx to scope our libraries, just like [npm scoped packages](#).

Next, you'll be prompted to select a **preset**—choose the `react` option.



```
macbook-pro:projects jack$ npx create-nx-workspace
✓ Workspace name (e.g., org name)      acme
? What to create in the new workspace ...
  empty                                [an empty workspace with a layout that works best for building apps]
> react                                [a workspace with a single React application]
  angular                              [a workspace with a single Angular application]
  next.js                              [a workspace with a single Next.js application]
  gatsby                               [a workspace with a single Gatsby application]
  nest                                  [a workspace with a single Nest application]
  express                              [a workspace with a single Express application]
  web components                       [a workspace with a single app built using web components]
  react-express                        [a workspace with a full stack application (React + Express)]
]
  angular-nest                        [a workspace with a full stack application (Angular + Nest)]
  oss                                  [an empty workspace with a layout that works best for open-source projects]
```

### Creating a workspace

After choosing the preset, you'll be prompted for the application name, and the styling format you want to use. Let's use `bookstore` as our application name and `styled-components` for styling.

```
macbook-pro:projects jack$ npx create-nx-workspace
✓ Workspace name (e.g., org name)      · acme
✓ What to create in the new workspace  · react
✓ Application name                     · demo
? Default stylesheet format            ...
CSS
SASS(.scss)      [ http://sass-lang.com   ]
LESS             [ http://lesscss.org   ]
Stylus(.styl)   [ http://stylus-lang.com ]
> styled-components [ https://styled-components.com ]
emotion         [ https://emotion.sh   ]
styled-jsx      [ https://www.npmjs.com/package/styled-jsx ]
```

### Choosing a style option

In addition, Nx asks about setting up Nx Cloud<sup>2</sup>. Nx Cloud adds remote distributed computation caching and other performance enhancing features to the Nx workspace. Even though it is the commercial add-on for Nx, it comes with a generous free tier. So feel free to go ahead and enable it or skip it entirely.

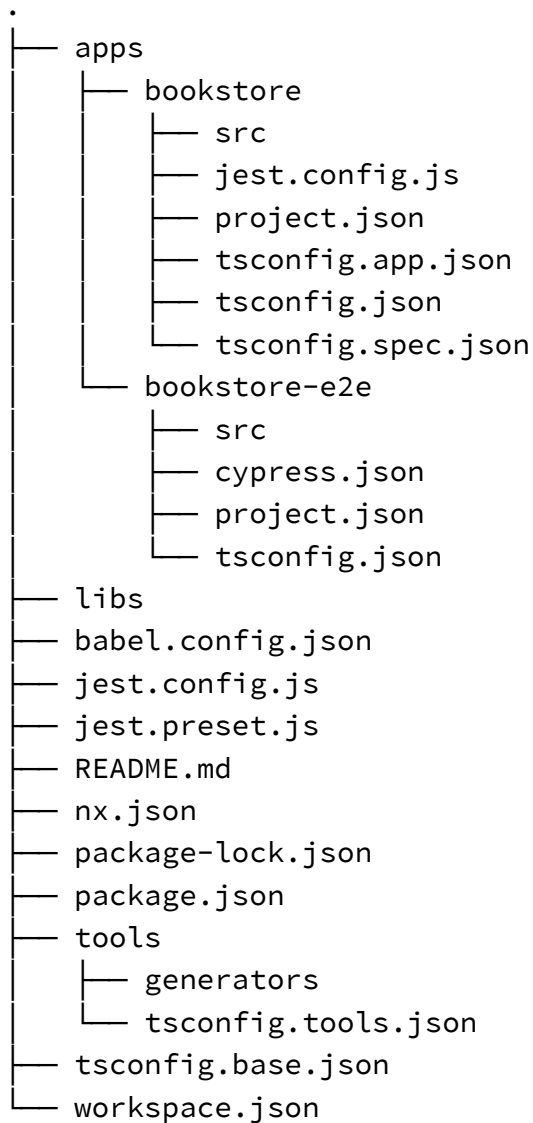


Note: If you prefer Yarn over npm, you can pass the `--packageManager=yarn` flag to the `create-nx-workspace`.

Once Nx finishes creating the workspace, we will end up with something like this:

---

<sup>2</sup><https://nx.app>



The `apps` folder contains the code of all applications in our workspace. Nx has created two applications by default:

- The `bookstore` application itself; and
- A set of end-to-end (E2E) tests written to test the `bookstore` application using Cypress<sup>3</sup>.

The `libs` folder will eventually contain our libraries (more on that in [Chapter 2](#)). It is empty for now.

---

<sup>3</sup><https://www.cypress.io/>

The `tools` folder can be used for scripts that are specific to the workspace. The generated `tools/generators` folder is for Nx's [workspace generators](#) feature which you can learn more about by reading the documentation at <https://nx.dev/generators/generators>.

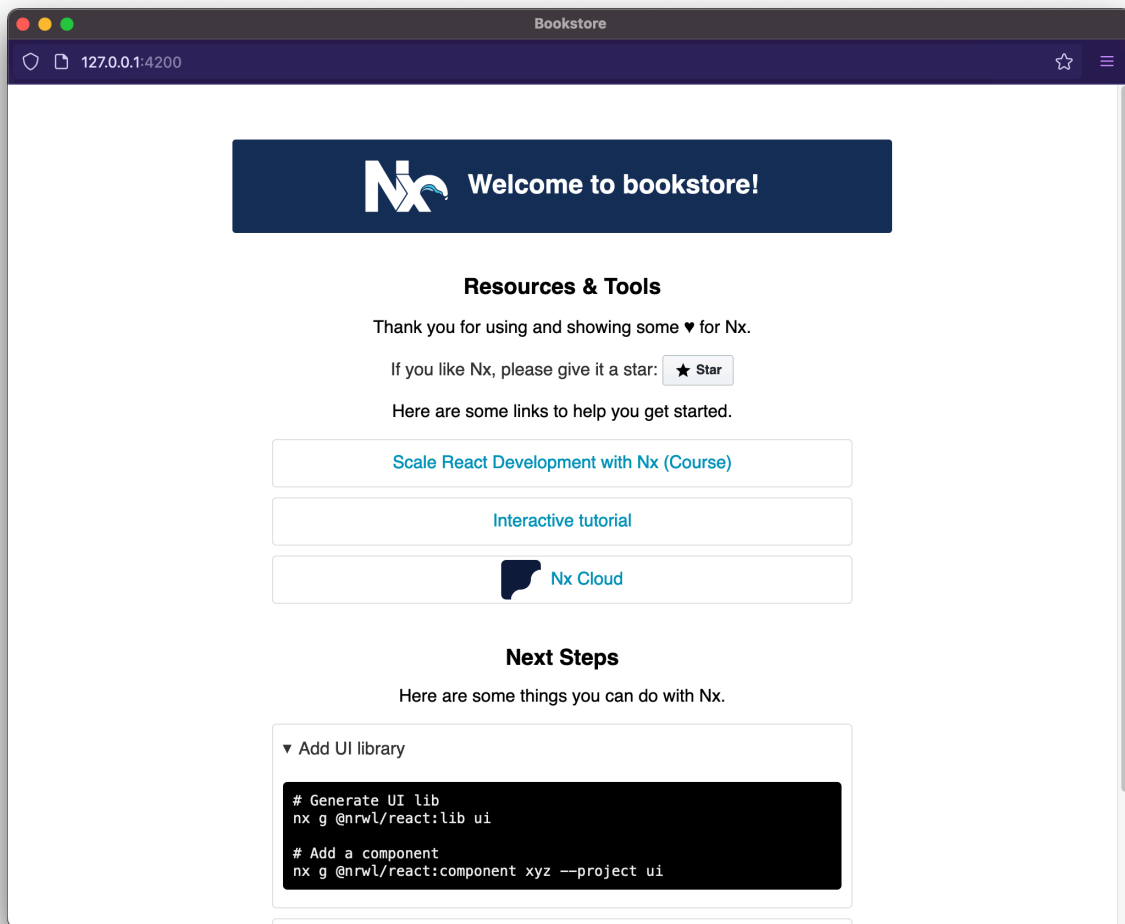
The `nx.json` file configures Nx. We're going to have a closer look at that in [Chapter 4](#).

To serve the application, use this command:

```
npm start
```

The above command uses the `start` script in the main `package.json` which builds the `bookstore` application and then starts a development server at port `4200`.

When we navigate to <http://localhost:4200> we are presented with a friendly welcome page.



The generated welcome page

## Nx workspace configuration

Nx is built in a modular fashion consisting of a core infrastructure that provides the foundation such as for the dependency graph calculation, running generators and migrations, computation caching and more, and a set of plugins that provide technology specific features (such as `@nrwl/react` for React development). Those plugins are developed and maintained by the Nx core team as well as the community<sup>4</sup>).

---

<sup>4</sup><https://nx.dev/community>

This allows to gradually dive deeper into the Nx features or simply to just start in a more lightweight fashion. You could easily just use the Nx core<sup>5</sup> and rely on other tooling such as Yarn/Npm workspaces to do the linking. Yet, you would miss out on a lot of features.

In this book we're going full-in. This allows us to explore all the features Nx can bring to the table when it comes to React development and thus set us up to be most productive. This setup comes with some configuration files that provide Nx with the necessary meta-data to be able to best reason about the structure of the underlying workspace. Let's briefly explore them in more detail.

The previously generated workspace comes with the follow Nx specific configuration files:

- `nx.json`
- `workspace.json`
- `project.json`

The `nx.json` is at the root of the workspace and configures the Nx CLI. It allows to specify things such as defaults for projects and code scaffolding, the workspace layout, task runner options and computation cache configuration and more. Here's an excerpt of what got generated for our example workspace.

```
{
  "npmScope": "acme",
  "affected": {
    "defaultBase": "main"
  },
  "cli": {
    "defaultCollection": "@nrwl/react"
  },
  "implicitDependencies": {
    "package.json": {
      "dependencies": "*",
      "devDependencies": "*"
    }
  }
}
```

---

<sup>5</sup><https://nx.dev/getting-started/nx-core>

```
    },
    ".eslintrc.json": "*"
  },
  "tasksRunnerOptions": {
    "default": {
      "runner": "@nrwl/workspace/tasks-runners/default",
      "options": {
        "cacheableOperations": ["build", "lint", "test", "e2e"]
      }
    }
  },
  "targetDependencies": {
    "build": [
      {
        "target": "build",
        "projects": "dependencies"
      }
    ]
  },
  "generators": {
    "@nrwl/react": {
      "application": {
        "style": "styled-components",
        "linter": "eslint",
        "babel": true
      },
      ...
    }
  },
  ...
}
```

The `workspace.json` file in the root directory is optional. It's used to list the projects in your workspace explicitly, instead of having Nx scan the file tree for all `project.json` and `package.json` files.

The `project.json` file is located at the root of every project in your workspace. This is where the project specific metadata is defined as well as the “targets”. A Nx target is literally a “task” that can be invoked on the project. Open the



apps/bookstore/project.json of the bookstore application:

```
{
  "root": "apps/bookstore",
  "sourceRoot": "apps/bookstore/src",
  "projectType": "application",
  "targets": {
    "build": { ... },
    "serve": { ... },
    "lint": { ... }
    "test": { ... }
  },
  ...
}
```

It contains targets for invoking a build, serve for serving the app during development as well as targets for linting (lint) and testing (test). These are the ones generated by default, but you are free to add your own as well.

Each of these targets comes with a set of things that can be configured. Let's have a look at the build target:

```
{
  ...
  "targets": {
    "build": {
      "executor": "@nrwl/web:webpack",
      "outputs": ["{options.outputPath}"],
      "options": {
        "compiler": "babel",
        "outputPath": "dist/apps/bookstore",
        "index": "apps/bookstore/src/index.html",
        "baseHref": "/",
        "main": "apps/bookstore/src/main.tsx",
        "polyfills": "apps/bookstore/src/polyfills.ts",
        "tsConfig": "apps/bookstore/tsconfig.app.json",
        "assets": [
          "apps/bookstore/src/favicon.ico",
```

```

    "apps/bookstore/src/assets"
  ],
  "styles": [],
  "scripts": [],
  "webpackConfig": "@nrwl/react/plugins/webpack"
},
"configurations": {
  "production": {
    "fileReplacements": [
      {
        "replace": "apps/bookstore/src/environments/environment.ts",
        "with": "apps/bookstore/src/environments/environment.prod.ts"
      }
    ],
    "optimization": true,
    "outputHashing": "all",
    "sourceMap": false,
    "namedChunks": false,
    "extractLicenses": true,
    "vendorChunk": false
  }
}
},
...
},
...
}

```

Each target comes with a Nx Executor<sup>6</sup> definition: `@nrwl/web:webpack`. An executor is a program (in this case named `webpack` and located in the `@nrwl/web` package) that is used to run the target. In this specific case it will use Webpack to create the application build. By abstracting the details of how the build is created into an executor, it takes away the burden of configuring Webpack and allows Nx to automatically handle Webpack upgrades and optimizations in an automated fashion, without breaking your workspace. That said, flexibility is still preserved. The executor comes with options to merge in your own Webpack

---

<sup>6</sup><https://nx.dev/executors/using-builders>

options and you can totally also create your own custom executor<sup>7</sup>.

The target comes also with options that are read by the executor to customize the outcome accordingly. Depending on the executor implementation the target is using, these options might vary.

Finally there's the configurations which extends the options and potentially overrides them with different values. This can be handy when building for different environments. Configurations can be activate by passing the `--configuration=<name>` flag to the command.

## Nx commands

As we mentioned in the previous section, targets can be invoked. You can curn them in the form: `nx [target] [project]`.

For example, for our bookstore app we can run the following targets.

```
# Serve the app
```

```
npx nx serve bookstore
```

```
# Build the app
```

```
npx nx build bookstore
```

```
# Run a linter for the application
```

```
npx nx lint bookstore
```

```
# Run unit tests for the application
```

```
npx nx test bookstore
```

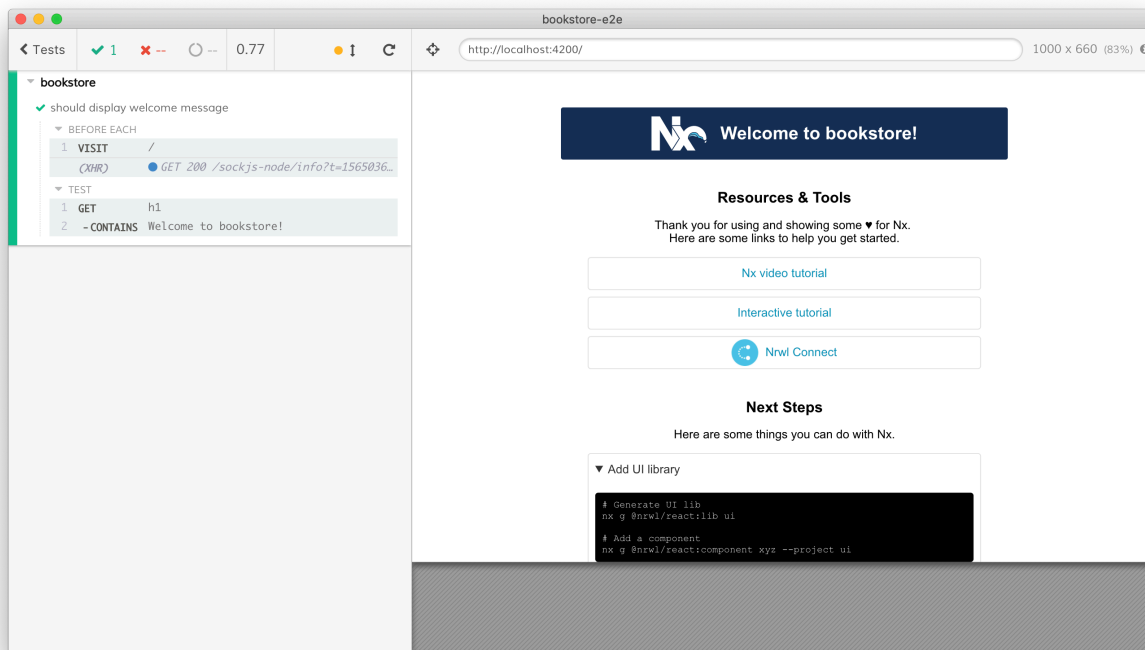
```
# Run e2e tests for the application
```

```
npx nx e2e bookstore-e2e
```

Give these commands a try!

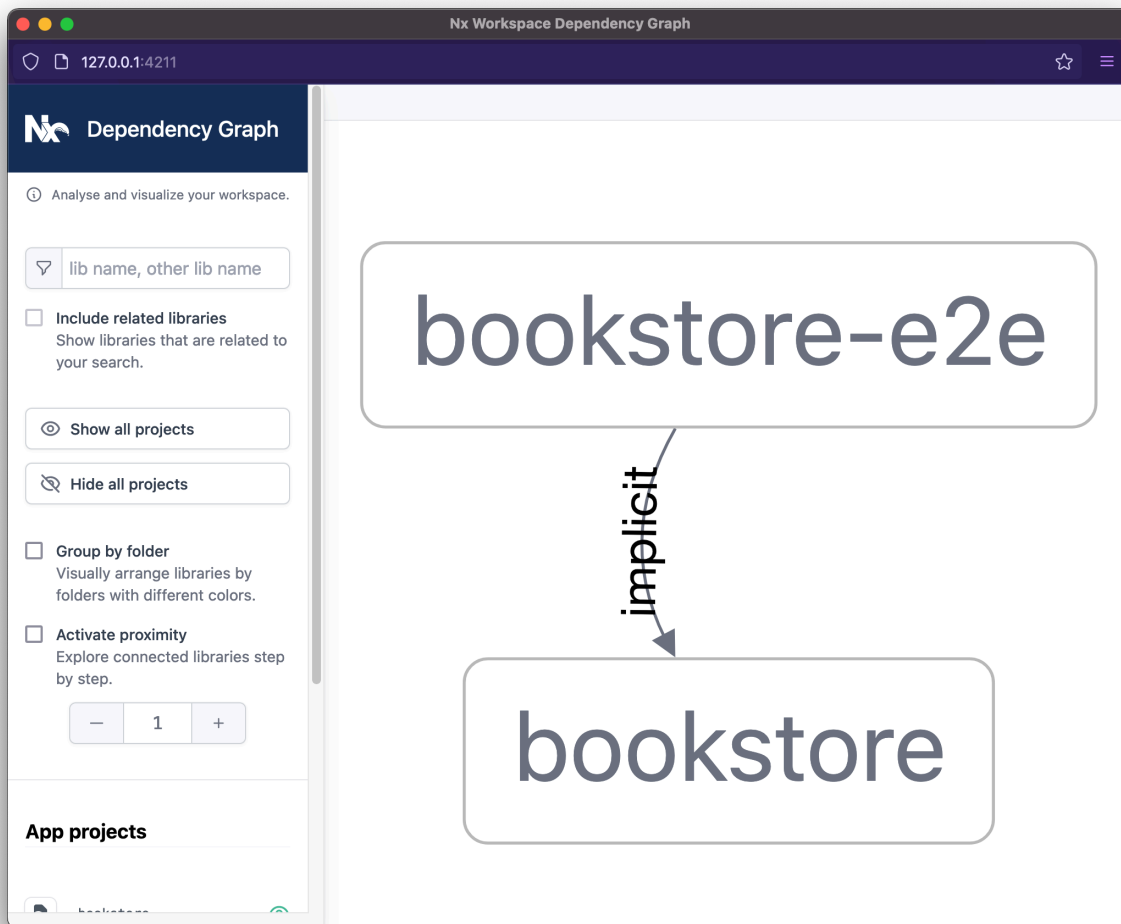
---

<sup>7</sup><https://nx.dev/executors/creating-custom-builders>



**`npm run nx e2e bookstore-e2e`**

Lastly, Nx allows us to examine the dependency graph of our workspace with the `npm run nx dep-graph` command.



Dependency graph of the workspace

There isn't much in the workspace to make this graph useful just yet, but we will see in later chapters how this feature can help us understand the architecture of our application, and how changes to code affect various projects within the workspace.

### Install Nx globally (optional)

It's easier to work with Nx when we have it installed globally. You can do this by running:

```
npm install -g @nrwl/cli
```

Check that installation has worked by issuing the command `nx --version`.

Now you will be able to run Nx commands without going through `npx` (e.g. `nx serve bookstore`).

For the rest of this book, I will assume that you have Nx installed globally. If you haven't, simply run all issued commands through `npx`.

## Preparing for development

Let's end this chapter by removing the generated content from the bookstore application and adding some configuration to the workspace.

Open up your favorite editor and modify these three files.

**apps/bookstore/src/app/app.tsx**

```
import styled from 'styled-components';
```

```
const StyledApp = styled.div``;
```

```
export const App = () => {
```

```
  return (
```

```
    <StyledApp>
```

```
      <header>
```

```
        <h1>Bookstore</h1>
```

```
      </header>
```

```
    </StyledApp>
```

```
  );
```

```
};
```

```
export default App;
```

**apps/bookstore/src/app/app.spec.tsx**

```
import { render, cleanup } from '@testing-library/react';

import App from './app';

describe('App', () => {
  afterEach(cleanup);

  it('should render successfully', () => {
    const { baseElement } = render(<App />);

    expect(baseElement).toBeTruthy();
  });

  it('should have a header as the title', () => {
    const { getByText } = render(<App />);

    expect(getByText('Bookstore')).toBeTruthy();
  });
});
```

### apps/bookstore-e2e/src/integration/app.spec.ts

```
import { getGreeting } from '../support/app.po';

describe('bookstore', () => {
  beforeEach(() => cy.visit('/'));

  it('should display welcome message', () => {
    getGreeting().contains('Bookstore');
  });
});
```

Make sure the tests still pass:

- nx lint bookstore
- nx test bookstore
- nx e2e bookstore-e2e

It's a good idea to commit our code before making any more changes.

```
git add .  
git commit -m 'end of chapter one'
```

---



### Key points

A typical Nx workspace consists of two types of projects: *applications* and *libraries*.

A newly created workspace comes with a set of targets we can run on the generated application: `lint`, `test`, and `e2e`.

Nx also has a tool for displaying the dependency graph of all the projects within the workspace.



# Chapter 2: Libraries

We have the skeleton of our application from [Chapter 1](#).

So now we can start adding to our application by creating and using libraries. Before we dive straight into creating libraries, though, let's first understand the concept of libraries in an Nx workspace.

## Apps and Libs

A typical Nx workspace is structured into “apps” and “libs”. This separation helps facilitate more modular architectures by following a separation of concerns methodology, incentivising the organisation of your source code and logic into smaller, more focused and highly cohesive units.

Nx automatically creates TypeScript path mappings in the `tsconfig.base.json`, such that they can be easily consumed by other apps or libs. More on that later.

```
// example of importing from another workspace library
import { Button } from '@acme/ui'
...
```

As a result, consuming libraries is very straightforward, and similar to what you might already be accustomed to in your current setup, where you structure code within folders of your React application project. Having a dedicated library project is a much stronger boundary compared to just separating code into folders, though. Each Nx library has a so-called “public API”, represented by a `index.ts` barrel file. This forces developers into an “API thinking” of what should be exposed and thus be made available for others to consume, and what on the others side should remain private within the library itself.



80% of the logic should reside in libraries, 20% in apps.

A common mental model is to see the application as “containers” that link, bundle and compile functionality implemented in libraries for being deployed. As such, if we follow a 80/20 approach: place 80% of your logic into the `libs/` folder, and 20% into `apps/`.

Note, these libraries don’t necessarily need to be built separately, but are rather consumed and built by the application itself directly. Hence, nothing changes from a pure deployment point of view. That said, it is totally possible to create so-called “buildable libraries” for enabling incremental builds<sup>8</sup> as well as “publishable libraries” for those scenarios where not only you want to use a specific library within the current Nx workspace, but also to publish it to some package repository (e.g NPM). You can read more about buildable and publishable libraries on the official Nx docs<sup>9</sup>.

## Organizing Libraries

Developers new to Nx are initially often hesitant to move their logic into libraries, because they assume it implies that those libraries need to be general purpose and shareable across applications. This is a common misconception. Moving code into libraries can be done from a pure code organization perspective.



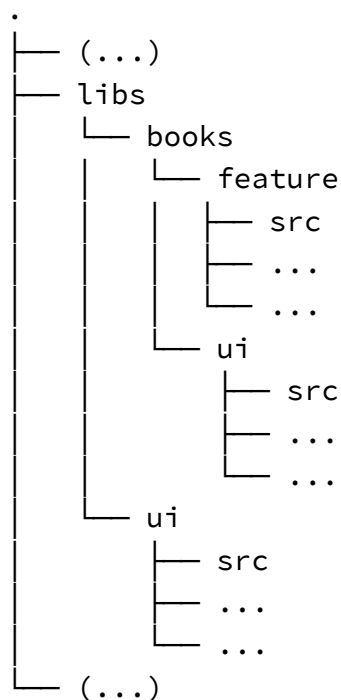
Ease of re-use might emerge as a positive side-effect of refactoring code into libraries by applying an “API thinking” approach. It is not the main driver though.

In fact when organizing libraries you should think about your business domains. Most often teams are aligned with those domains and thus a similar organization of the libraries in the `libs` folder might be most appropriate. Nx allows to nest libraries into sub-folders which makes it easy to reflect such structuring.

---

<sup>8</sup><https://nx.dev/ci/incremental-builds>

<sup>9</sup><https://nx.dev/structure/buildable-and-publishable-libraries>



Note how we might have a `libs/books/feature` and `libs/books/ui` library, both of which are specific libraries for the `books` domain, while `libs/ui` represents a more general purpose library of UI elements such as a common UI design system that can be used across all of the other domains. Applying such a nested structure can be powerful to organize your workspace as well as for applying code ownership rules as we'll see later.

## Categories of libraries

In a workspace, libraries are typically divided into four different types:

### Feature

Libraries that implement “smart” UI (e.g. is effectful, is connected to data sources, handles routing, etc.) for specific **business use cases**.

**UI** Libraries that contain only **presentational** components. That is, components that render purely from their props, and calls function handlers when interaction occurs.

### Data-access

Libraries that contain the means for interacting with external data services; external services are typically backend services.

### Utility

Libraries that contain common utilities that are shared by many projects.

Why do we make these distinctions between libraries? Good question! It is good to set boundaries for what a library should and should not do. This demarcation makes it easier to understand the capabilities of each library, and how they interact with each other.

More concretely, we can form rules about what each types of libraries can depend on. For example, UI libraries cannot use feature or data-access libraries, because doing so will mean that they are effectful.

We'll see in later in this chapter how we can use Nx to strictly enforce these boundaries.

## The generate command

The `nx generate` or the `nx g` command, as it is aliased, allows us to use Nx generator to create new applications, components, libraries, and more, to our workspace.

## Feature libraries

Let's create our first feature library: books.

```
nx g lib feature \
--directory books \
--appProject bookstore \
--tags type:feature,scope:books
```

The `--directory` option allows us to group our libraries by nesting them under their parent directory. In this case the library is created in the `libs/books/feature` folder. It is aliased to `-d`.

The `--appProject` option lets Nx know that we want to make our feature library to be routable inside the specified application. This option is useful because Nx will do three things for us.

The `--tags` option lets us annotate our applications and libraries to express constraints within the workspace. The tags are added to `project.json`, and we'll see at the end of this chapter how they can be used to enforce different constraints.

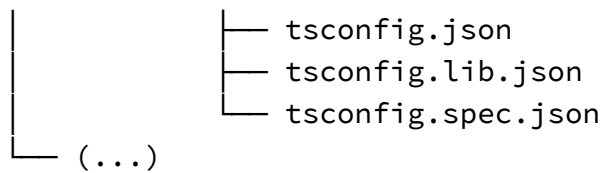
1. Update `apps/bookstore/src/app/app.tsx` with the new route.
2. Update `apps/bookstore/src/main.tsx` to add `BrowserRouter` if it does not exist yet.
3. Add `react-router-dom` and related dependencies to the workspace, if necessary.



**Pro-tip:** You can pass the `--dryRun` option to `generate` to see the effects of the command before committing to disk.

Once the command completes, you should see the new directory.

```
.
├── (...)
├── libs
│   └── books
│       └── feature
│           ├── src
│           │   ├── index.ts
│           │   └── lib
│           ├── jest.config.js
│           ├── project.json
│           └── README.md
```



Nx generated our library with some default code as well as scaffolding for linting (ESLint) and testing (Jest). You can run them with:

```
nx lint books-feature
nx test books-feature
```

You'll also see that the App component for bookstore has been updated to include the new route.

```
import styled from 'styled-components';

import { Route, Link } from 'react-router-dom';

import { BooksFeature } from '@acme/books/feature';
const StyledApp = styled.div``;
export const App = () => {
  return (
    <StyledApp>
      <header>
        <h1>Bookstore</h1>
      </header>

      {/* START: routes */}
      {/* These routes and navigation have been generated for you */}
      {/* Feel free to move and update them to fit your needs */}
      <br />
      <hr />
      <br />
      <div role="navigation">
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
```

```
        <li>
          <Link to="/feature">BooksFeature</Link>
        </li>
        <li>
          <Link to="/page-2">Page 2</Link>
        </li>
      </ul>
    </div>
    <Route
      path="/"
      exact
      render={() => (
        <div>
          This is the generated root route.{' '}
          <Link to="/page-2">Click here for page 2.</Link>
        </div>
      )}
    />
    <Route path="/feature" component={BooksFeature} />
    <Route
      path="/page-2"
      exact
      render={() => (
        <div>
          <Link to="/">Click here to go back to root page.</Link>
        </div>
      )}
    />
    {/* END: routes */}
  </StyledApp>
);
};
export default App;
```

Additionally, the `main.tsx` file for bookstore has also been updated to render `<BrowserRouter />`. This render is needed in order for `<Route />` components to work, and Nx will handle the file update for us if necessary.

```
import { StrictMode } from 'react';
import * as ReactDOM from 'react-dom';

import App from './app/app';

import { BrowserRouter } from 'react-router-dom';

ReactDOM.render(
  <StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </StrictMode>,
  document.getElementById('root')
);
```

Restart the development server again (`nx serve bookstore`), and you should see the updated application.



Be aware when you add a new project to the workspace, you must restart your development server. This restart is necessary in order for the TypeScript compiler to pick up new library paths, such as `@acme/books/feature`.

By using a monorepo, we've *skipped* a few steps that are usually required when creating a new library.

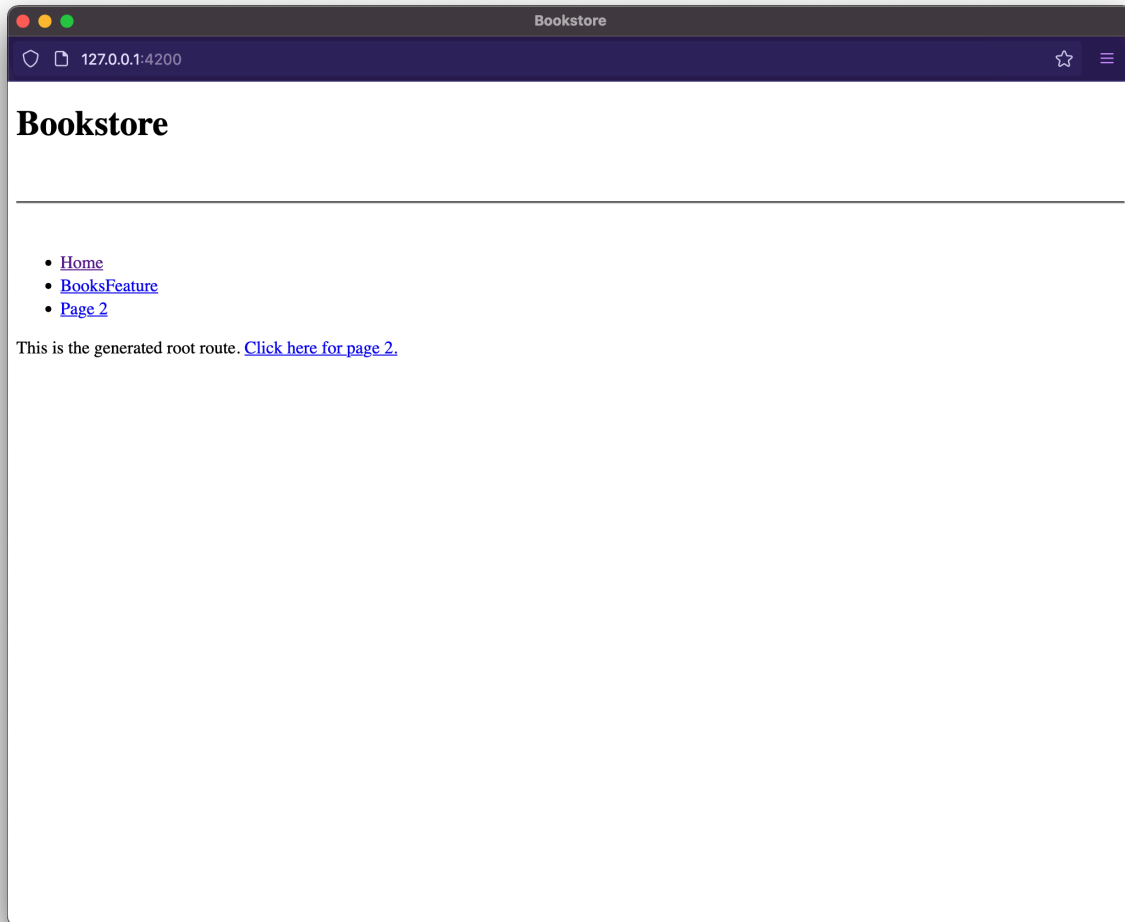
- Setting up the repo
- Setting up the CI
- Setting up the publishing pipeline—such as artifactory

And now we have our library! Wasn't that easy? Something that may have taken minutes or hours—sometimes even days—now takes only takes a few seconds.

---



But to our despair, when we navigate to <http://localhost:4200> again, we see a poorly styled application.



Let's remedy this situation by adding a component library that will provide better styling.

## UI libraries

Let's create the UI library.

```
nx g lib ui \  
--tags type:ui,scope:books \  
--no-interactive
```

The `--no-interactive` tells Nx to not prompt us with options, but instead use the default values.



Please note that we will make heavy use of [styled-components](#) in this component library. Don't fret if you're not familiar with styled-components. If you know CSS then you should not have a problem understanding this section. To learn more about styled-components you can check out their [documentation](#).

You should have a new folder: `libs/ui`.

```
acme  
├── (...)  
├── libs  
│   ├── (...)  
│   └── ui  
│       ├── src  
│       │   ├── lib  
│       │   └── index.ts  
│       ├── .eslintrc  
│       ├── jest.config.js  
│       ├── README.md  
│       ├── tsconfig.app.json  
│       ├── tsconfig.json  
│       └── tsconfig.spec.json  
└── (...)
```

This library isn't quite useful yet, so let's add in some components.

```
nx g component GlobalStyles --project ui --export --tags type:ui,scope:books
nx g component Button --project ui --export --tags type:ui,scope:books
nx g component Header --project ui --export --tags type:ui,scope:books
nx g component Main --project ui --export --tags type:ui,scope:books
nx g component NavigationList --project ui --export --tags type:ui,scope:books
nx g component NavigationItem --project ui --export --tags type:ui,scope:books
```

The `--project` option specifies which project (as found in the `projects` section of `workspace.json`) to add the new component to. It is aliased to `-p`.

The `--export` option tells Nx to export the new component in the `index.ts` file of the project so that it can be imported elsewhere in the workspace. You may leave this option off if you are generating private/internal components. It is aliased to `-e`.

If you do forget the `--export` option you can always manually add the export barrel to `index.ts`.



**Pro-tip:** There are additional options and aliases available to the `nx g component` command. To see a list of options run `nx g component --help`. Also, check out `nx g lib --help` and `nx g app --help`!

Next, let's go over the implementation of each of the components and what their purposes are.

## GlobalStyles

This component injects a global stylesheet into our application when used. It is particularly useful for overriding global style rules such as `body { margin: 0 }`.

**libs/ui/src/lib/global-styles/global-styles.tsx**

```
import { createGlobalStyle } from 'styled-components';

export const GlobalStyles = createGlobalStyle`
  body {
    margin: 0;
    font-size: 16px;
    font-family: sans-serif;
  }

  * {
    box-sizing: border-box;
  }
`;

export default GlobalStyles;
```

## Button

This component is pretty self-explanatory. It renders a styled button and passes through other props to the actual `<button>`.

### libs/ui/src/lib/button/button.tsx

```
import { ButtonHTMLAttributes } from 'react';
import styled from 'styled-components';

const StyledButton = styled.button`
  font-size: 0.8rem;
  padding: 0.5rem;
  border: 1px solid #ccc;
  background-color: #fafafa;
  border-radius: 4px;

  &:hover {
    background-color: #80a8e2;
    border-color: #0e2147;
  }
`;
```

```
`;  
  
export const Button = ({  
  children,  
  ...rest  
}: ButtonHTMLAttributes<HTMLButtonElement>) => {  
  return <StyledButton {...rest}>{children}</StyledButton>;  
};  
  
export default Button;
```

## Header and Main

These two components are used for layout. The header component forms the top header bar, while the main component takes up the rest of the page.

### libs/ui/src/lib/header/header.tsx

```
import { HTMLAttributes } from 'react';  
import styled from 'styled-components';  
  
const StyledHeader = styled.header`  
  padding: 1rem;  
  background-color: #2657ba;  
  color: white;  
  display: flex;  
  align-items: center;  
  
  a {  
    color: white;  
    text-decoration: none;  
  
    &:hover {  
      text-decoration: underline;  
    }  
  }  
`
```

```
> h1 {
  margin: 0 1rem 0 0;
  padding-right: 1rem;
  border-right: 1px solid white;
}
`;

export const Header = (props: HTMLAttributes<HTMLElement>) => (
  <StyledHeader>{props.children}</StyledHeader>
);

export default Header;
```

### libs/ui/src/lib/main/main.tsx

```
import { HTMLAttributes } from 'react';
import styled from 'styled-components';

const StyledMain = styled.main`
  padding: 0 1rem;
  width: 100%;
  max-width: 960px;
`;

export const Main = (props: HTMLAttributes<HTMLElement>) => (
  <StyledMain>{props.children}</StyledMain>
);

export default Main;
```

### NavigationList and NavigationItem

And finally, the `NavigationList` and `NavigationItem` components will render the navigation bar inside our top `Header` component.

### libs/ui/src/lib/navigation-list/navigation-list.tsx

```
import { HTMLAttributes } from 'react';
import styled from 'styled-components';

const StyledNavigationList = styled.div`
  ul {
    display: flex;
    margin: 0;
    padding: 0;
    list-style: none;
  }
`;

export const NavigationList = (props: HTMLAttributes<HTMLElement>) => {
  return (
    <StyledNavigationList role="navigation">
      <ul>{props.children}</ul>
    </StyledNavigationList>
  );
};

export default NavigationList;
```

### libs/ui/src/lib/navigation-item/navigation-item.tsx

```
import { LiHTMLAttributes } from 'react';
import styled from 'styled-components';

const StyledNavigationItem = styled.li`
  margin-right: 1rem;
`;

export const NavigationItem = (props: LiHTMLAttributes<HTMLLIElement>) => {
  return <StyledNavigationItem>{props.children}</StyledNavigationItem>;
};

export default NavigationItem;
```

## Using the UI library

Now we can use the new library in our bookstore's app component.

**apps/bookstore/src/app/app.tsx**

```
import { Link, Redirect, Route } from 'react-router-dom';

import { BooksFeature } from '@acme/books/feature';

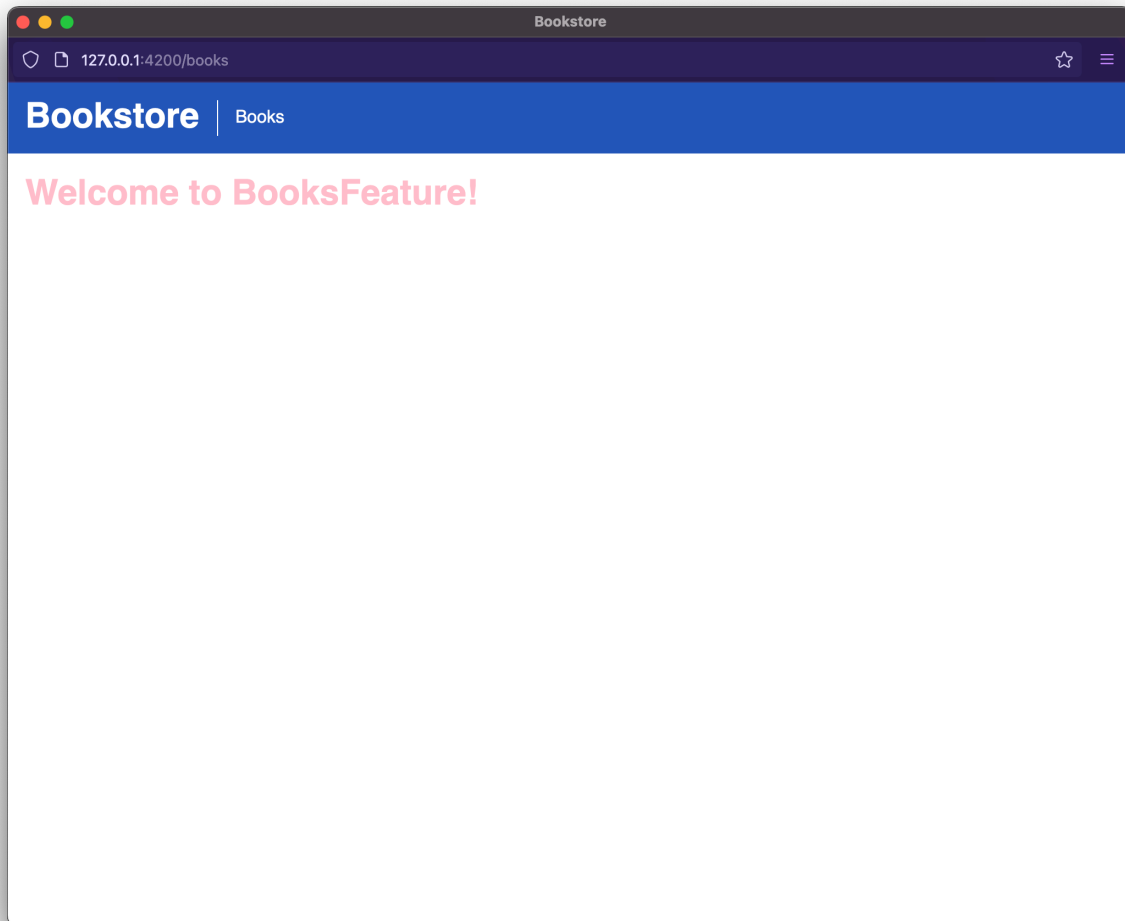
// importing the UI library into our App
import {
  GlobalStyles,
  Header,
  Main,
  NavigationItem,
  NavigationList
} from '@acme/ui';

export const App = () => {
  return (
    <>
      <GlobalStyles />
      <Header>
        <h1>Bookstore</h1>
        <NavigationList>
          <NavigationItem>
            <Link to="/books">Books</Link>
          </NavigationItem>
        </NavigationList>
      </Header>
      <Main>
        <Route path="/books" component={BooksFeature} />
        <Route exact path="/" render={() => <Redirect to="/books" />} />
      </Main>
    </>
  );
};
```



```
export default App;
```

Finally, let's restart our server (`nx serve bookstore`) and we will see a much improved UI.



We'll save our progress with a new commit.

```
git add .  
git commit -m 'add books feature and ui components'
```

That's great, but we are still not seeing any books, so let's do something about this.

## Data-access libraries

What we want to do is fetch data from *somewhere* and display that in our books feature. Since we will be calling a backend service we should create a new **data-access** library.

```
nx g @nrwl/web:lib data-access --directory books --tags type:data-access,scope:books
```

You may have noticed that we are using a prefix `@nrwl/web:lib` instead of just `lib` like in our previous examples. This `@nrwl/web:lib` syntax means that we want Nx to run the `lib` (or `library`) generator provided by the `@nrwl/web` collection.

We were able to go without this prefix previously because the `nx.json` configuration has set `@nrwl/react` as the default option.

```
{
  // ...
  "cli": {
    "defaultCollection": "@nrwl/react"
  },
  // ...
}
```

In this case, the `@nrwl/web:lib` generator will create a library to be used in a web (i.e. browser) context without assuming the framework used. In contrast, when using `@nrwl/react:lib`, it assumes that you want to generate a default component as well as potentially setting up routes.



**Pro-tip:** A collection in Nx contains a set of *generators* and *executors*. Generators can be invoked using the `generate` command. Executors perform actions on your code, including build, lint, and test and are invoked by issuing commands to Nx—such as `nx lint` and `nx test`. Use `nx list [collection]` to list everything provided by the collection—e.g. `nx list @nrwl/react`.

Back to the example. Let's modify the library to export a `getBooks` function to load our list of books.

### `libs/books/data-access/src/lib/books-data-access.ts`

```
export async function getBooks() {  
  // TODO: We'll wire this up to an actual API later.  
  // For now we are just returning some fixtures.  
  return [  
    {  
      id: 1,  
      title: 'The Picture of Dorian Gray',  
      author: 'Oscar Wilde',  
      rating: 3,  
      price: 9.99  
    },  
    {  
      id: 2,  
      title: 'Frankenstein',  
      author: 'Mary Wollstonecraft Shelley',  
      rating: 5,  
      price: 7.95  
    },  
    {  
      id: 3,  
      title: 'Jane Eyre',  
      author: 'Charlotte Brontë',  
      rating: 4,  
      price: 10.95  
    },  
    {  
      id: 4,  
      title: 'Dracula',  
      author: 'Bram Stoker',  
      rating: 5,  
      price: 14.99  
    },  
    {  
      id: 5,
```

```
    title: 'Pride and Prejudice',  
    rating: 4,  
    author: 'Jane Austen',  
    price: 12.85  
  }  
];  
}
```

## Using the data-access library

The next step is to use the `getBooks` function within our `books` feature. We can do this with React's `useEffect` and `useState` hooks.

### `libs/books/feature/src/lib/books-feature.tsx`

```
import { useEffect, useState } from 'react';  
import styled from 'styled-components';  
import { getBooks } from '@acme/books/data-access';  
import { Books } from '@acme/books/ui';  
  
export const BooksFeature = () => {  
  const [books, setBooks] = useState<any[]>([]);  
  
  useEffect(() => {  
    getBooks().then(setBooks);  
  }, [  
    // This effect runs only once on first component render  
    // so we declare it as having no dependent state.  
  ]);  
  
  return (  
    <>  
      <h2>Books</h2>  
      <Books books={books} />  
    </>  
  );  
};
```

```
export default BooksFeature;
```

You'll notice that we're using two new components: `Books` and `Book`. They can be created as follows.

```
nx g lib ui --directory books
nx g component Books --project books-ui --export
nx g component Book --project books-ui --export
```

We generally want to put *presentational* components into their own UI library. This will prevent side-effects from bleeding into them, thus making them easier to understand and test.



Note how we generate a new `books-ui` library under `libs/books/ui` rather than using the already existing `ui` library under `libs/ui`. The reason is that the former contains specific presentational components for the books feature of our workspace, while the latter contains the general purpose UI components that form our corporate design system components.

Again, we will see later in this chapter how Nx enforces module boundaries.

### `libs/books/ui/src/lib/books/books.tsx`

```
import styled from 'styled-components';
import { Book } from '../book/book';

export interface BooksProps {
  books: any[];
}

const StyledBooks = styled.div`
  border: 1px solid #ccc;
  border-radius: 4px;
`;

export const Books = ({ books }: BooksProps) => {
```

```
    return (  
      <StyledBooks>  
        {books.map(book => (  
          <Book key={book.id} book={book} />  
        ))}  
      </StyledBooks>  
    );  
  };  
};
```

```
export default Books;
```

### libs/books/ui/src/lib/book/book.tsx

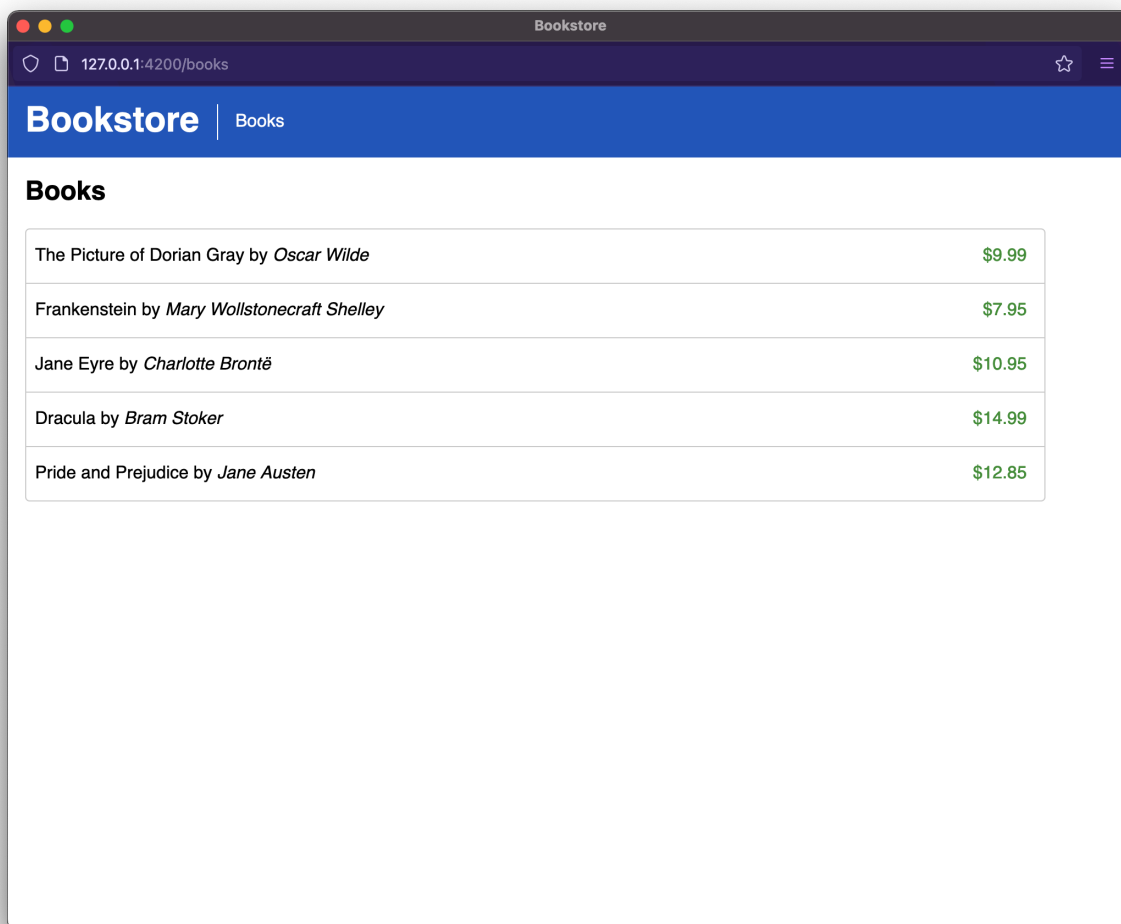
```
import styled from 'styled-components';  
import { Button } from '@acme/ui';
```

```
export interface BookProps {  
  book: any;  
}
```

```
const StyledBook = styled.div`  
  display: flex;  
  align-items: center;  
  border-bottom: 1px solid #ccc;  
  &:last-child {  
    border-bottom: none;  
  }  
  > span {  
    padding: 1rem 0.5rem;  
    margin-right: 0.5rem;  
  }  
  .title {  
    flex: 1;  
  }  
  .price {  
    color: #478d3c;  
  }  
`;  
;
```

```
export const Book = ({ book }: BookProps) => {  
  return (  
    <StyledBook>  
      <span className="title">  
        {book.title} by <em>{book.author}</em>  
      </span>  
      <span className="price">${book.price}</span>  
    </StyledBook>  
  );  
};  
  
export default Book;
```

Restart the server to check out our feature in action.



That’s great and all, but you may have observed a couple of problems.

1. The `getBooks` data-access function is a stub and doesn’t actually call out to a backend service.
2. We’ve been using `any` types when dealing with books data. For example, the return type of `getBooks` is `any[]` and our `BookProp` takes specifies `{ book: any }`. This makes our code unsafe and can lead to production bugs.

We’ll address both problems in the [next chapter](#). For now, let’s wrap up by examining how Nx can enforce module boundaries between different types of libraries that we’ve created in this chapter.



## Enforcing module boundaries

Recall that earlier, when we generated our libraries we passed the `--tags` option to define a *type* and *scope* for each of them. Let's examine how we can use these tags to define and enforce clean separation of concerns within the workspace.

Open up the `.eslintrc.json` file at the root of your workspace. It will contain an entry for `@nrwl/nx/enforce-module-boundaries` as follows.

```
"@nrwl/nx/enforce-module-boundaries": [
  "error",
  {
    "depConstraints": [
      {
        "sourceTag": "*",
        "onlyDependOnLibsWithTags": ["*"]
      }
    ],
    "allow": [],
    "enforceBuildableLibDependency": true
  }
]
```

The `depConstraints` section is the one you will be spending most time fine-tuning. It is an array of constraints, each consisting of `sourceTag` and `onlyDependOnLibsWithTags` properties. The default configuration has a wildcard `*` set as a value for both of them, meaning that any project can import (depend on) any other project.

The circular dependency chains such as `lib A -> lib B -> lib C -> lib A` are also not allowed. The self circular dependency (when `lib` imports from a named alias of itself), while not recommended, can be overridden by setting the flag `allowCircularSelfDependency` to `true`.

```
"@nrwl/nx/enforce-module-boundaries": [  
  "error",  
  {  
    "allowCircularSelfDependency": true,  
    ...  
  }  
]
```

The `allow` array is a whitelist listing the import definitions that should be omitted from further checks. We will see how overrides work after we define the `depConstraints` section.

Finally, the flag `enforceBuildableLibDependency` prevents us from importing a non-buildable library into a buildable one.

## Using tags to enforce boundaries

Earlier in this chapter we presented [four categories of libraries](#) we typically find in a Nx workspace.

1. Feature
2. UI
3. Data-access
4. Utility

We've already added these types as tags to our libraries when we ran the `generate` command (e.g. `--tags type:ui`). We also want to consider a fifth type of project in the workspace: `type:app` that we can tag all of our applications with.

Now, let's define some constraints for what each types of projects can depend on.

- Applications can depend on any types of libraries, but not other applications.

- Feature libraries can depend on any other library.
- UI libraries can only depend on other UI or utility libraries.
- Utility libraries can only depend on other utility libraries.

Let's see how we can configure the ESLint rule to enforce the above constraints.

```
"@nrwl/nx/enforce-module-boundaries": [
  "error",
  {
    ...
    "depConstraints": [
      {
        "sourceTag": "type:app",
        "onlyDependOnLibsWithTags": ["type:feature", "type:ui", "type:util"]
      },
      {
        "sourceTag": "type:feature",
        "onlyDependOnLibsWithTags": ["type:feature", "type:ui", "type:util"]
      },
      {
        "sourceTag": "type:ui",
        "onlyDependOnLibsWithTags": ["type:ui", "type:util"]
      },
      {
        "sourceTag": "type:util",
        "onlyDependOnLibsWithTags": ["type:util"]
      }
    ]
  }
]
```

Further, recall that we also have a second dimension to the tags of our libraries (e.g. `--tag scope:books`). The `scope` tag allows us to separate our applications and libraries into logical domains.

Imagine that we want to add an **admin** application to our workspace in order to manage our books. We can create such application with `nx g app admin`

`--tags type:app,scope:admin`. It is likely that there will be some shared libraries required by both **admin** and **books** applications.

For example, if we have common components—such as buttons, modals, etc.—then we can generate a shared library as follows.

```
nx g lib ui --directory shared --tags type:ui,scope:shared
```

And we want to enforce the following rules for scopes.

- Book application and libraries can only depend on `scope:books` libraries.
- Admin application and libraries can only depend on `scope:admin` libraries.
- Any applications and libraries can depend on `scope:shared` libraries.

This is how we would define the constraints.

```
"@nrwl/nx/enforce-module-boundaries": [
  "error",
  {
    ...
    "depConstraints": [
      {
        "sourceTag": "type:app",
        "onlyDependOnLibsWithTags": ["type:feature", "type:ui", "type:util"]
      },
      ...
      {
        "sourceTag": "scope:books",
        "onlyDependOnLibsWithTags": ["scope:shared", "scope:books"]
      },
      {
        "sourceTag": "scope:admin",
        "onlyDependOnLibsWithTags": ["scope:shared", "scope:admin"]
      },
      {
        "sourceTag": "scope:shared",
```

```
    "onlyDependOnLibsWithTags": ["scope:shared"]
  }
]
}
```

By forbidding cross-scope dependencies we can prevent a feature from `admin` being used in the `books` application. Also, domain-specific logic can be safely guarded against being used in the wrong domains.

These module boundaries are needed as the workspace grows, otherwise projects will become unmanageable, and changes will be hard to reason about. You can customize the tags however you want. If you have a multi-platform monorepo, you might add `platform:web`, `platform:node`, `platform:native`, and `platform:all` tags.

Learn more about configuring boundaries with the [Taming Code Organization with Module Boundaries in Nx](#) article<sup>10</sup>.

Finally, let's commit our changes up to this point before moving on to the next chapter.

```
git add .
git commit -m 'implement books feature and link to application'
```

---

<sup>10</sup><https://blog.nrwl.io/mastering-the-project-boundaries-in-nx-f095852f5bf4>



### Key points

There are four type of libraries: *feature*, *UI*, *data-access*, and *util*.

Nx provides us with the `nx generate` or `nx g` command to *quickly* create new libraries from scratch.

When running `nx g` we can optionally provide a collection such as `@nrwl/web:lib` as opposed to `lib`. This will tell Nx to use the generator from that specific collection rather than the default as set in `nx.json`.

Nx enforces module boundaries through tags and `@nrwl/nx/enforce-module-boundaries` ESLint rule. The boundaries allow us to better organize and manage our workspace.

# Chapter 3: Working effectively in a monorepo

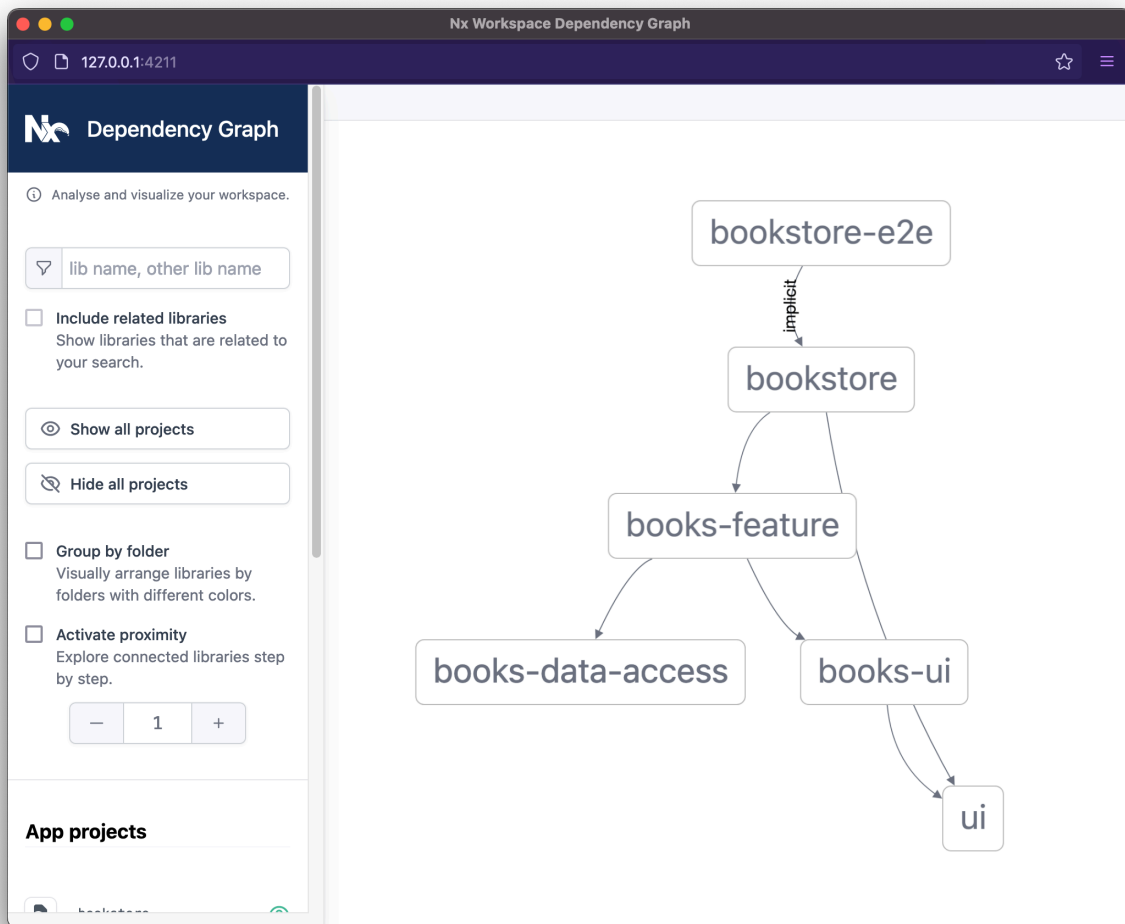
In the previous two chapters we set up a bookstore application that renders a list of books for users to purchase.

In this chapter we explore how Nx enables us to work more effectively.

## The dependency graph

As we've seen in [Chapter 1](#), Nx automatically generates the dependency graph for us. So why don't we see how it looks now?

```
nx dep-graph
```



Dependency graph of the workspace

Nx knows the dependency graph of the workspace without us having to configure anything. Because of this ability, Nx also understands which projects within the workspace are affected by any given changeset. Moreover, it can help us verify the correctness of the affected projects.



Note that you can also manually add so-called “implicit dependencies” for those rare cases where there needs to be a dependency which can not be automatically inferred from source code. Read more about that here: <https://nx.dev/configuration/projectjson#implicitdependencies>



## Only recompute affected projects

Let's say we want to add a **checkout** button to each of the books in the list.

We can update our `Book`, `Books`, and `BooksFeature` components to pass along a new `onAdd` callback prop.

`libs/books/ui/src/lib/book/book.tsx`

```
import styled from 'styled-components';
import { Button } from '@acme/ui';
```

```
export interface BookProps {
  book: any;
  // New prop
  onAdd: (book: any) => void;
}
```

```
const StyledBook = styled.div`
  display: flex;
  align-items: center;
  border-bottom: 1px solid #ccc;
  &:last-child {
    border-bottom: none;
  }
  > span {
    padding: 1rem 0.5rem;
    margin-right: 0.5rem;
  }
  .title {
    flex: 1;
  }
  .rating {
    color: #999;
  }
  .price {
    color: #478d3c;
  }
`
```

```

`;

export const Book = ({ book, onAdd }: BookProps) => {
  const handleAdd = () => onAdd(book);
  return (
    <StyledBook>
      <span className="title">
        {book.title} by <em>{book.author}</em>
      </span>
      <span className="rating">{book.rating}</span>
      <span className="price">${book.price}</span>
      {/* Add button to UI */}
      <span>
        <Button onClick={handleAdd}>Add to Cart</Button>
      </span>
    </StyledBook>
  );
};

export default Book;

```

### libs/books/ui/src/lib/books/books.tsx

```

import styled from 'styled-components';
import { Book } from '../book/book';

export interface BooksProps {
  books: any[];
  // New prop
  onAdd: (book: any) => void;
}

const StyledBooks = styled.div`
  border: 1px solid #ccc;
  border-radius: 4px;
`;

export const Books = ({ books, onAdd }: BooksProps) => {
  return (

```

```
    <StyledBooks>
      {books.map(book => (
        // Pass down new callback prop
        <Book key={book.id} book={book} onAdd={onAdd} />
      ))}
    </StyledBooks>
  );
};
```

```
export default Books;
```

### libs/books/feature/src/lib/books-feature.tsx

```
import { useEffect, useState } from 'react';
import styled from 'styled-components';
import { getBooks } from '@acme/books/data-access';
import { Books, Book } from '@acme/books/ui';

export const BooksFeature = () => {
  const [books, setBooks] = useState<any[]>([]);

  useEffect(() => {
    getBooks().then(setBooks);
  }, [
    // This effect runs only once on first component render
    // so we declare it as having no dependent state.
  ]);

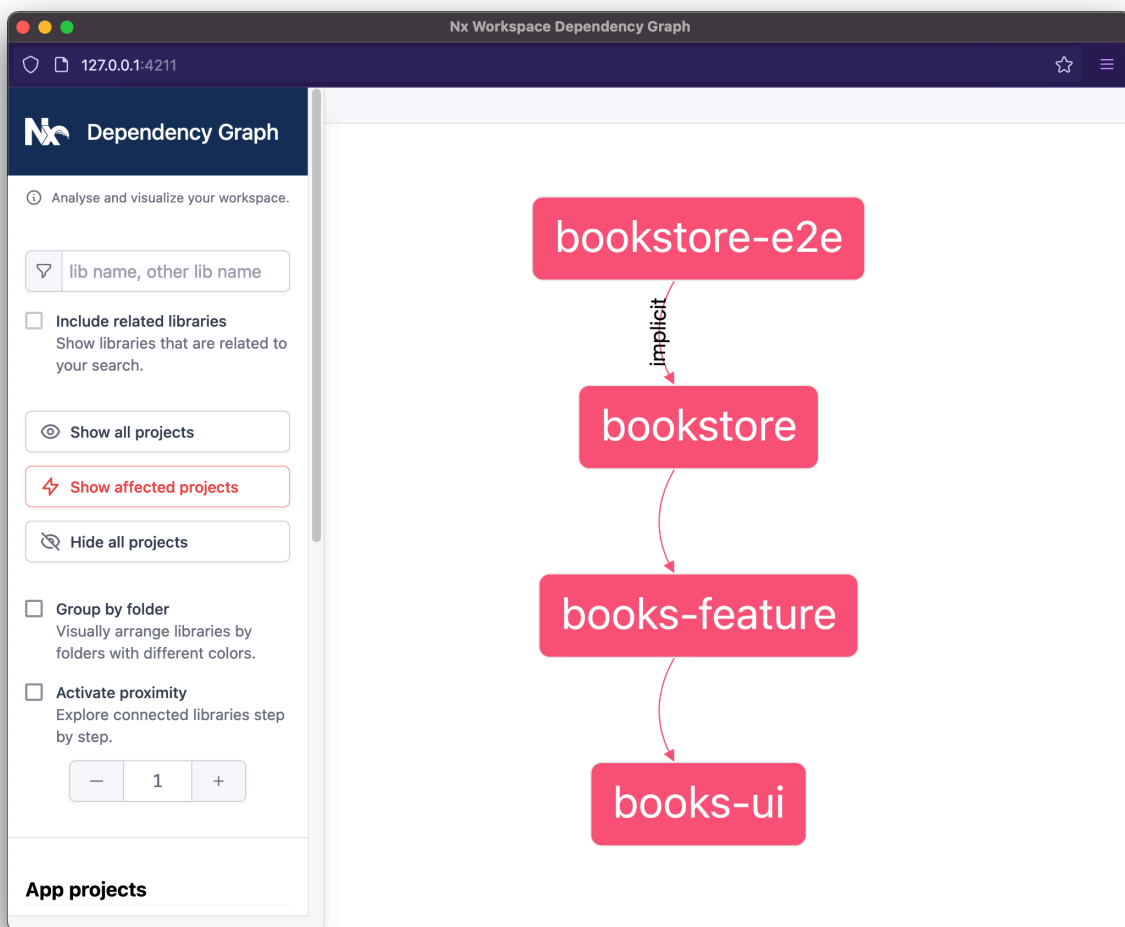
  return (
    <>
      <h2>Books</h2>
      {/* Pass a stub callback for now */}
      {/* We'll implement this properly in Chapter 4 */}
      <Books books={books} onAdd={book => alert(`Added ${book.title}`)} />
    </>
  );
};

export default BooksFeature;
```

By leveraging the dependency graph, Nx is not only able to understand how the workspace projects relate to each other, but combining this with the Git history, Nx is able to determine which projects were affected by a given changeset.

We can ask Nx to show us how this change *affects* the projects within our workspace using the so-called “affected command”.

```
nx affected:dep-graph
```



### Affected dependencies

As we can see, Nx knows that the `books-ui` library has changed starting from the Git `main` branch. Using this information, Nx walks up the dependency graph and highlights all the dependent projects affected by this change in *red*.

But there is more. We can not only just visualize this change, but we can use various commands to run only against this affected set of projects. Hence, we can just re-test, re-lint or re-build what changed.

```
// build only the affected apps
nx affected:build

// run unit tests on affected projects
nx affected:test

// run linting on affected projects
nx affected:lint

// run e2e tests on affected projects
nx affected:e2e
```

Nx topologically sorts the projects such that they are run from bottom to top. That is, projects at the bottom of the dependency chain run first. All these tasks are also parallelized by default (you can customize the amount of parallel tasks using `--maxParallel`).



Nx uses 3 parallel tasks by default. You can customize the amount using the `--maxParallel` flag.

All of the `affected:*` commands use the Git history, comparing the current HEAD with a “base” to determine which Nx project(s) got changed. By default “base” refers to the `main` branch. You can customize that by either passing the `--base` flag to the command or by changing the `defaultBase` property in `nx.json`.

Note that in these projects, Nx is using [Jest](#) and [Cypress](#) to run unit and e2e tests respectively. They make writing and running tests are fast and simple as possible. If you're not familiar with them, please read their documentation to learn more.

It is possible to use different executors by specifying them in the project's

```
project.json configuration file.
```

So far we haven't been diligent about verifying that our changes are okay, so unsurprisingly our tests are failing.

I'll leave it to you as an exercise to fix the broken unit and e2e tests. A hint for the App component test, you should look into the `MemoryRouter` from `React Router`.



**Pro-Tip:** You can run a target for an individual project by issuing `nx [target] [project]` such as `nx test books-ui`, `nx test bookstore`, or `nx e2e bookstore-e2e`. You may also pass the `--watch` flag to re-run tests as soon there is a code change.

For the full solution please see the bookstore example repository: <https://github.com/jaysoo/react-book-example>.

There are some additional affected commands in Nx.

1. `nx affected:apps` - Lists out all applications affected by the changeset.
2. `nx affected:libs` - Lists out all libraries affected by the changeset.

The listing of affected applications and libraries can be useful in CI to trigger downstream jobs based on the output.

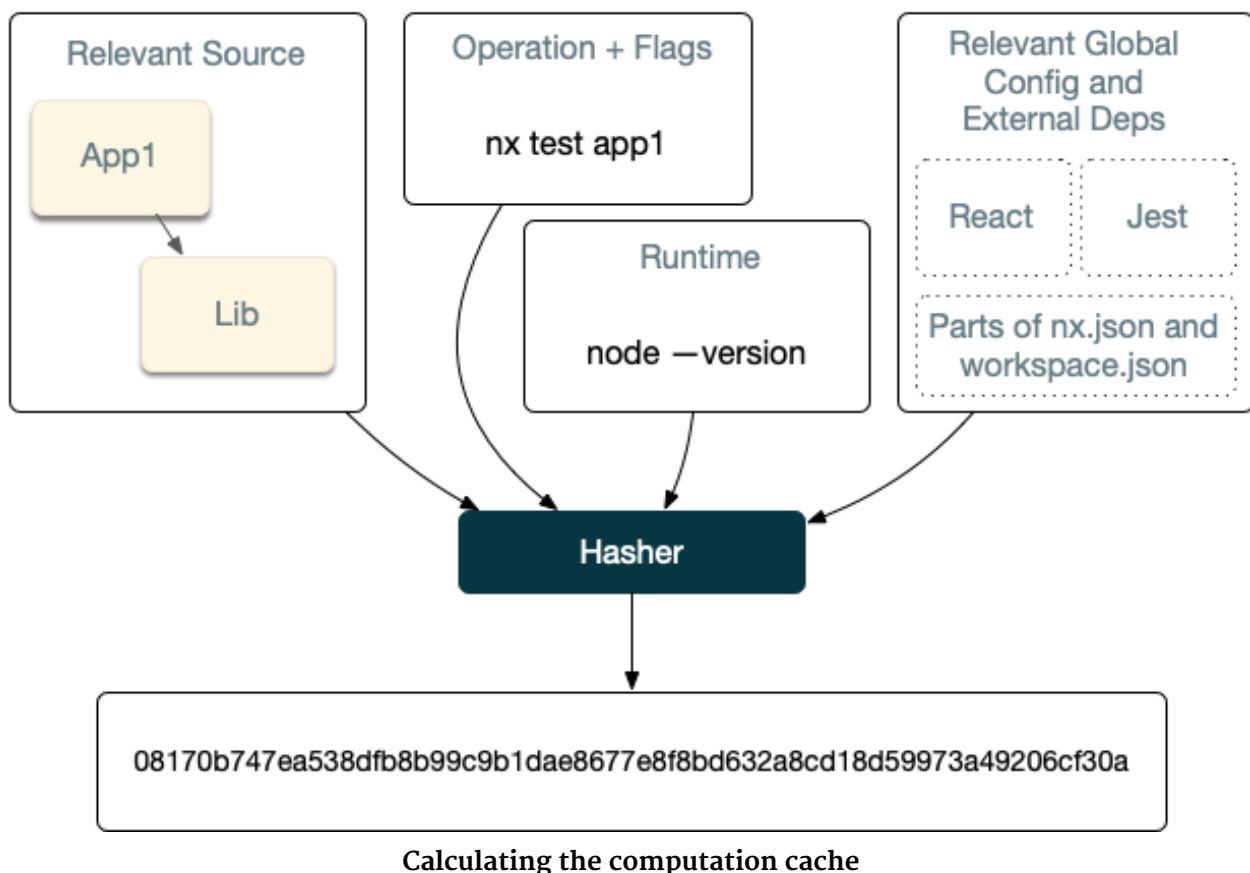
## Computation Caching

When you heavily adopt a monorepo and the number of projects grows, you need to start thinking about scaling. We have already seen how Nx can cut down the amount of projects to recompute drastically by using the previously mentioned “affected” commands.

Nx goes a step further by also using a computation cache. Basically before running any task, Nx calculates its computation hash. As long as the hash is the same, the output of running the task will be the same.

Let's take the example of running a unit test for an application `app1`. By default the computation hash includes

- All the source files of `app1` and its dependencies
- Relevant global configuration
- Versions of external dependencies
- Runtime values provisioned by the user such as the version of Node
- CLI Command flags



While this is the default behavior, it can also be customized to more specific needs. For instance, lint checks may only depend on the source code of the project and global configs. Or similarly, builds may depend on the dts files of the compiled libs instead of their source.

Once Nx has the computation hash, it verifies whether that specific hash already exists in its cache. If it does, it replays the task's output in the terminal and restores all possible files in the right folders. From a developers perspective it looks like the task was just run, simply a lot faster.

Try it out by yourself by running unit tests for the `books-feature` project. Run it once and then again to see it being restored from the cache the 2nd time.

```
// run unit tests
nx test books-feature

// run them again, they should be restored from the cache
nx test books-feature
```

Every Nx workspace has the computation caching enabled by default. Nx stores the cache locally in the `node_modules/.cache/nx` folder. You can customize which operations get cached as well as the exact location of the cache folder in the `nx.json` file under the `taskRunnerOptions` field.

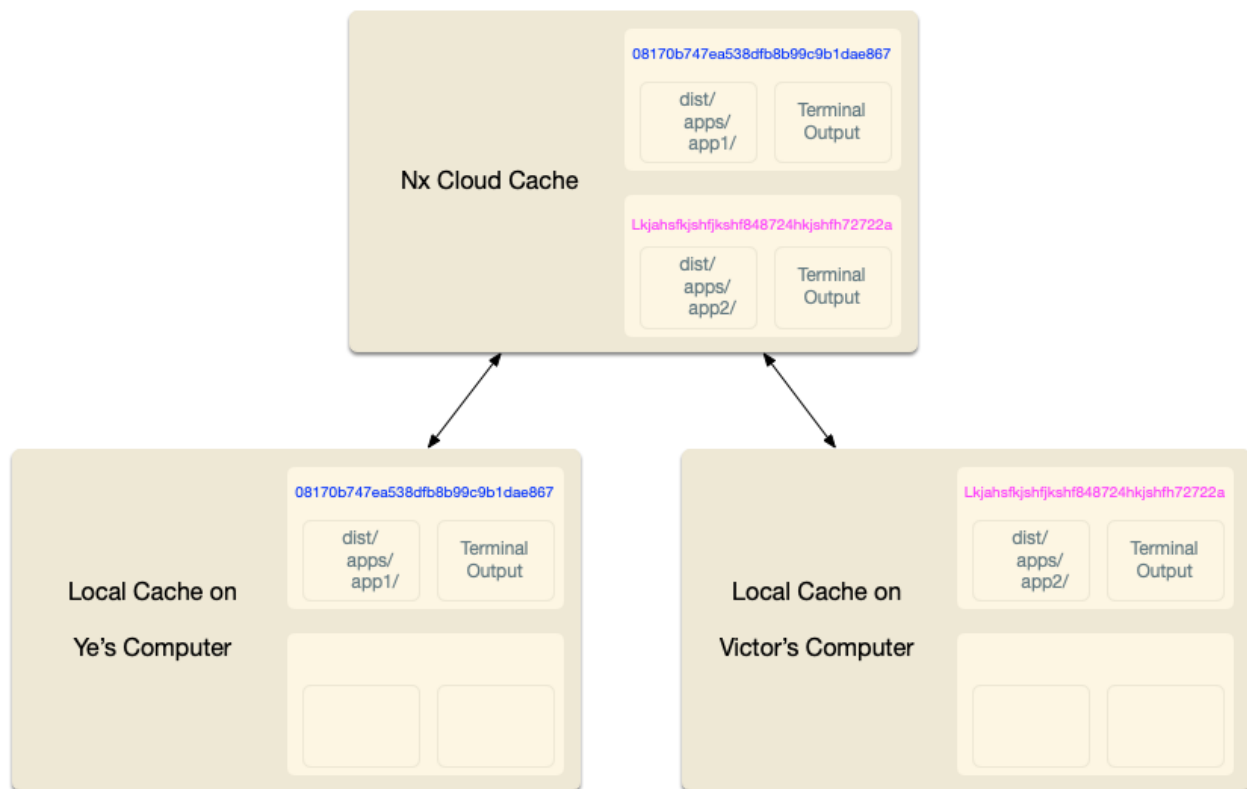
```
{
  ...
  "taskRunnerOptions": {
    "default": {
      "runner": "@nrwl/workspace/tasks-runners/default",
      "options": {
        "cacheableOperations": ["build", "lint", "test", "e2e"]
      }
    }
  },
}
```

You can get even more benefits if this cache is not only local, but remotely distributed. Such functionality can be enabled by using Nx Cloud<sup>11</sup>.

---

<sup>11</sup><https://nx.app>





Remote caching with Nx Cloud

If Nx Cloud is enabled, the local cache folder will be synced with a cloud-hosted, remote counterpart. With the remote cache, other team members and CI agents can read from it too and drastically reduce the required computation time. Learn more on <https://nx.app> and the corresponding Nx Cloud docs at <https://nx.app/docs>.

## Adding the API application

It's time to get more practical again and commit your changes if you haven't done so already: `git add . ; git commit -m 'added checkout button'`.

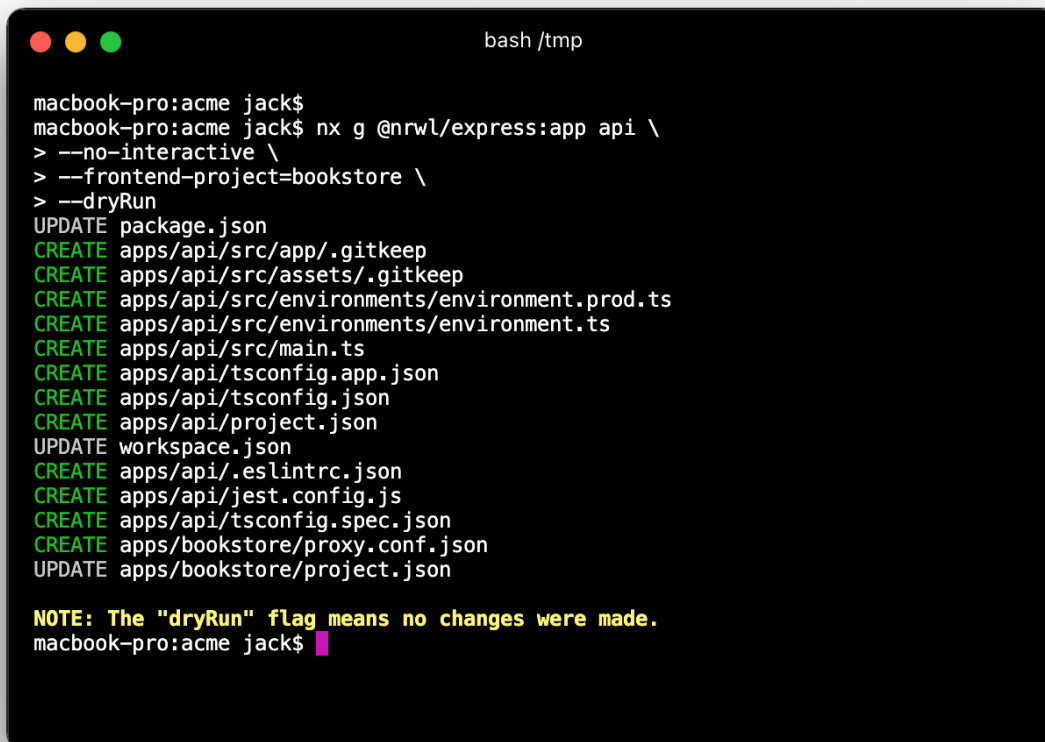
So far our bookstore application does not communicate with a real backend service. Let's create one using the [Express](#) framework.

We'll need to install the `@nrwl/express` collection first.

```
npm install --save-dev @nrwl/express
```

Then we can do a dry run of the express app generator.

```
nx g @nrwl/express:app api \  
--no-interactive \  
--frontend-project=bookstore \  
--dryRun
```



```
bash /tmp  
macbook-pro:acme jack$  
macbook-pro:acme jack$ nx g @nrwl/express:app api \  
> --no-interactive \  
> --frontend-project=bookstore \  
> --dryRun  
UPDATE package.json  
CREATE apps/api/src/app/.gitkeep  
CREATE apps/api/src/assets/.gitkeep  
CREATE apps/api/src/environments/environment.prod.ts  
CREATE apps/api/src/environments/environment.ts  
CREATE apps/api/src/main.ts  
CREATE apps/api/tsconfig.app.json  
CREATE apps/api/tsconfig.json  
CREATE apps/api/project.json  
UPDATE workspace.json  
CREATE apps/api/.eslintrc.json  
CREATE apps/api/jest.config.js  
CREATE apps/api/tsconfig.spec.json  
CREATE apps/bookstore/proxy.conf.json  
UPDATE apps/bookstore/project.json  
  
NOTE: The "dryRun" flag means no changes were made.  
macbook-pro:acme jack$
```

### Preview of the file changes

Everything looks good so let's run it for real.

```
nx g @nrwl/express:app api \
--no-interactive \
--frontend-project=bookstore
```

The `--frontend-project` option will add a proxy configuration to the bookstore application such that requests going to `/api/*` will be forwarded to the API

Just like our frontend application, we can use Nx to serve the API.

```
nx serve api
```

When we open up `http://localhost:3333/api` we'll be greeted by a friendly message.

```
{ "message": "Welcome to api!" }
```

Next, let's implement the `/api/books` endpoint so that we can use it in our `books-data-access` library.

### **apps/api/src/main.ts**

```
import * as express from 'express';

const app = express();

app.get('/api', (req, res) => {
  res.send({ message: 'Welcome to api!' });
});

app.get('/api/books', (req, res) => {
  const books: any[] = [
    {
      id: 1,
      title: 'The Picture of Dorian Gray ',
      author: 'Oscar Wilde',
      rating: 5,
      price: 9.99
    }
  ]
});
```

```
    },
    {
      id: 2,
      title: 'Frankenstein',
      author: 'Mary Wollstonecraft Shelley',
      rating: 4,
      price: 7.95
    },
    {
      id: 3,
      title: 'Jane Eyre',
      author: 'Charlotte Brontë',
      rating: 4.5,
      price: 10.95
    },
    {
      id: 4,
      title: 'Dracula',
      author: 'Bram Stoker',
      rating: 4,
      price: 14.99
    },
    {
      id: 5,
      title: 'Pride and Prejudice',
      author: 'Jane Austen',
      rating: 4.5,
      price: 12.85
    }
  ];
  res.send(books);
});

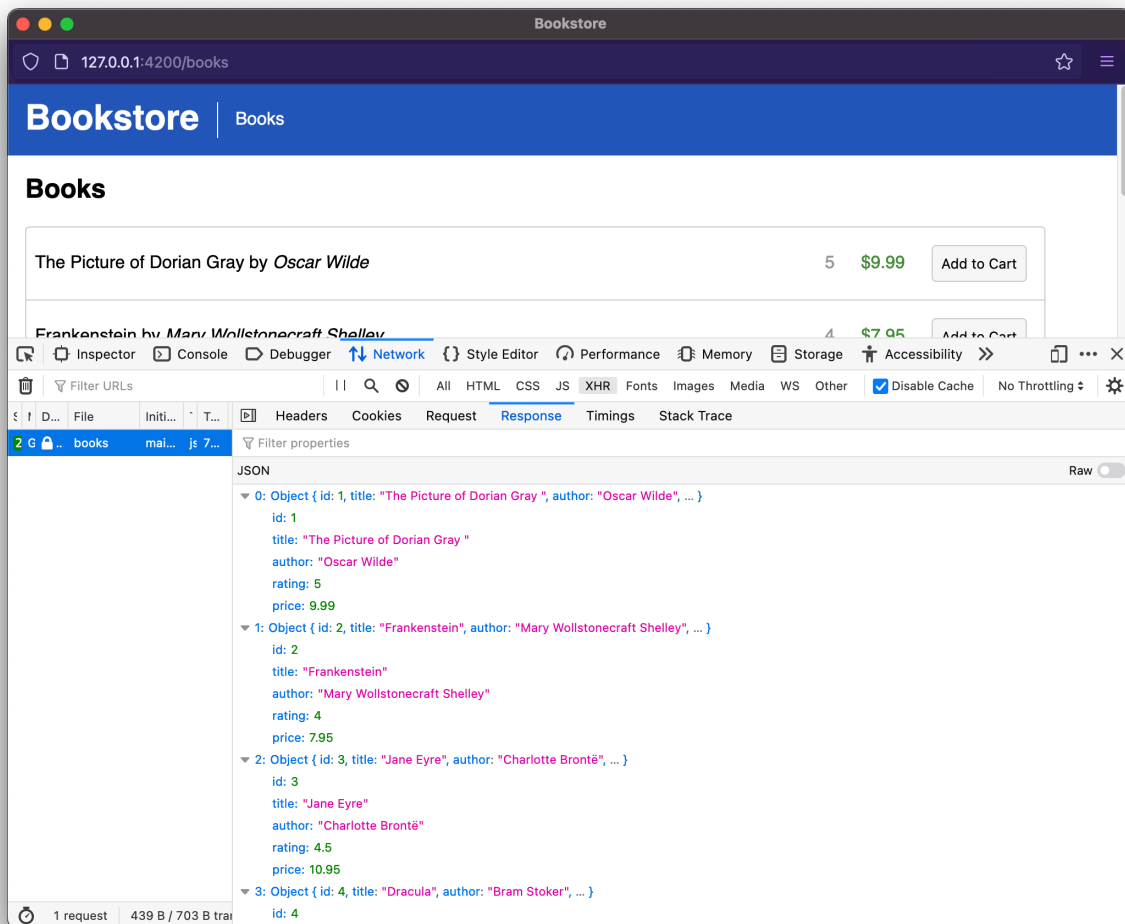
const port = process.env.port || 3333;
const server = app.listen(port, () => {
  console.log(`Listening at http://localhost:${port}/api`);
});
server.on('error', console.error);
```

Let's update our data-access library to call the proxy endpoint.

**libs/books/data-access/src/lib/books-data-access.ts**

```
export async function getBooks() {
  const data = await fetch('/api/books', {
    headers: {
      'Content-Type': 'application/json'
    }
  });
  return data.json();
}
```

If we restart both applications (`nx serve api` and `nx serve bookstore`; or in a single command `nx run-many --target=serve --projects=api,bookstore`) we'll see that our **bookstore** is still working in the browser. Moreover, we can verify that our `/api/books` endpoint is indeed being called.



Let's commit our changes: `git add . ; git commit -am 'added api app'`.

## Sharing models between frontend and backend

Recall that we previously used the `any` type when working with books data. This is bad practice as it may lead to uncaught type errors in production.

A better idea would be to create a utility library containing some shared models to be used by both the frontend and backend.

```
nx g @nrwl/node:lib shared-models --no-interactive
```

**libs/shared-models/src/lib/shared-models.ts**

```
export interface IBook {  
  id: number;  
  title: string;  
  author: string;  
  rating: number;  
  price: number;  
}
```

And now we can update the following five files to use the new model:

### apps/api/src/main.ts

```
import { IBook } from '@acme/shared-models';  
// ...  
  
app.get('/api/books', (req, res) => {  
  const books: IBook[] = [  
    // ...  
  ];  
  res.send(books);  
});  
  
// ...
```

### libs/books/data-access/src/lib/books-data-access.ts

```
import { IBook } from '@acme/shared-models';  
  
// Add correct type for the return value  
export async function getBooks(): Promise<IBook[]> {  
  const data = await fetch('http://localhost:3333/api/books');  
  return data.json();  
}
```

### libs/books/feature/src/lib/books-feature.tsx

```
...
import { IBook } from '@acme/shared-models';

export const BooksFeature = () => {
  // Properly type the array
  const [books, setBooks] = useState<IBook[]>([]);

  // ...

  return (
    <>
      <h2>Books</h2>
      <Books books={books} onAdd={book => alert(`Added ${book.title}`)} />
    </>
  );
};

export default BooksFeature;
```

### libs/books/ui/src/lib/books/books.tsx

```
// ...
import { IBook } from '@acme/shared-models';

// Replace any with IBook
export interface BooksProps {
  books: IBook[];
  onAdd: (book: IBook) => void;
}

// ...

export default Books;
```

### libs/books/ui/src/lib/book/book.tsx

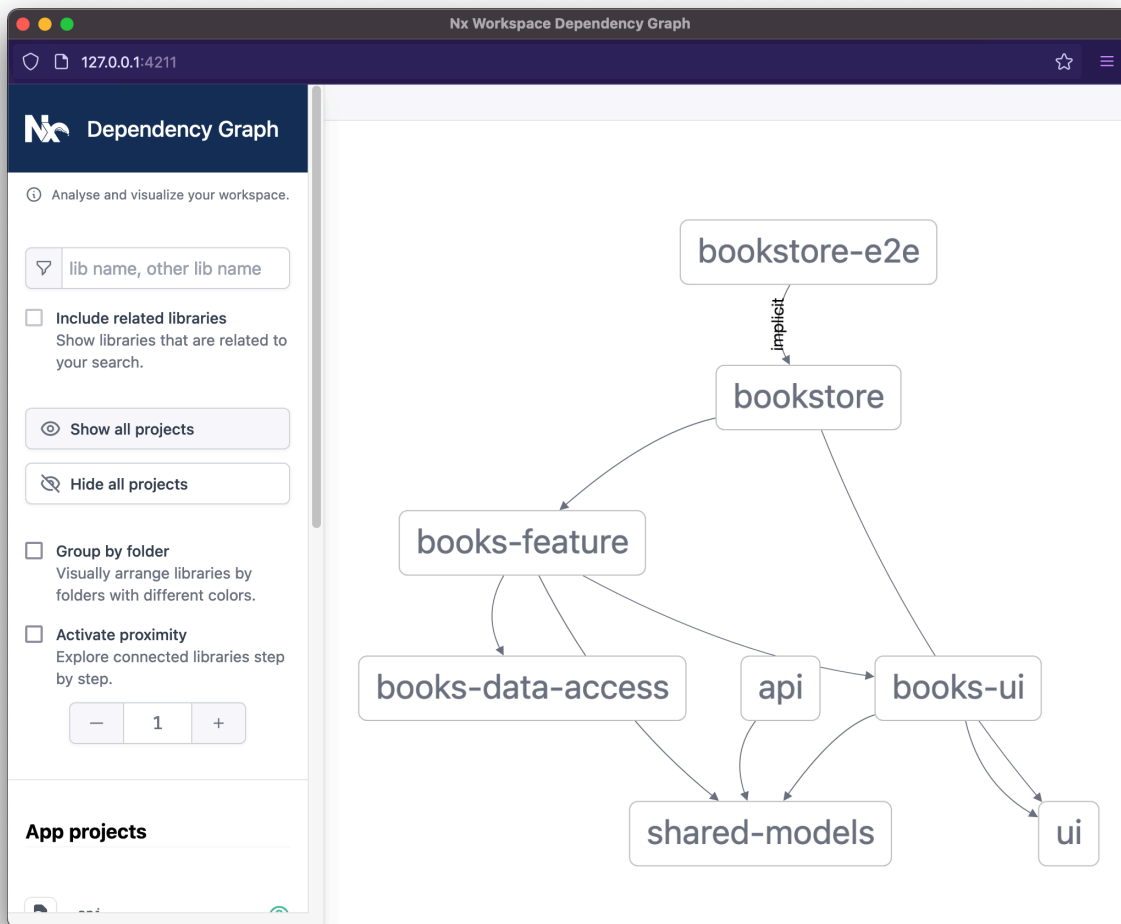


```
// ...
import { IBook } from '@acme/shared-models';

// Replace any with IBook
export interface BookProps {
  book: IBook;
  onAdd: (book: IBook) => void;
}

// ...

export default Book;
```



dependency graph with `api` and `shared-models`

By using Nx, we have created a shared model library and refactored both frontend and backend code in about a minute.

Another major benefit of working within a monorepo is that we can check in these changes as a *single commit*: `git add . ; git commit -m 'add shared models'`. The corresponding pull-request with the commit will have the full story, rather than being fragmented amongst multiple pull-requests and repositories.

## Automatic code formatting

One of the easiest ways to waste time as a developer is on code style. We can spend *hours* debating with one another on whether we should use semicolons or not (you should); or whether we should use a comma-first style or not (you should not).

**Prettier** was created to stop these endless debates over code style. It is highly opinionated and provides minimal configuration options. Best of all, it can format our code *automatically*. This means that we no longer need to manually fix code to conform to the code style.

Nx workspaces come with Prettier installed from the get-go. With it, we can check the formatting of the workspace, and format workspace code automatically.

```
# Checks for format conformance with Prettier.  
# Exits with error code when the check fails.  
nx format:check  
  
# Formats files with Prettier.  
nx format:write
```



### Key points

Nx understands the dependency graph of projects within our workspace.

We can ask Nx to generate the dependency graph automatically, as well as highlight the parts of the graph that are affected by a given changeset.

Nx can retest and rebuild only the affected projects within our workspace.

By using a monorepo, related changes in different projects can be in the same changeset (i.e. pull-request), which gives us the full picture of the changes.

Nx automatically formats our code for us in an opinionated way using Prettier.

# Chapter 4: Bringing it all together

Thus far in this book we've seen how to generate and organize our React application, libraries, and an Express application. In this chapter we will apply what we've learned so far to implement the checkout feature. We will also discuss how we can provide interaction between features through the use of React context and reducer.

## Checkout API and shared models

Let's add the new shared models for our shopping cart.

**libs/shared-models/src/lib/shared-models.ts**

```
// ...  
  
export interface ICartItem {  
  id: number;  
  description: string;  
  cost: number;  
}  
  
export interface ICart {  
  items: ICartItem[];  
}
```

Next, we can add a checkout endpoint to the API application.

**apps/api/src/main.ts**

```
import { IBook, ICart } from '@acme/shared-models';
// ...

app.post('/api/checkout', (req, res) => {
  const cart: ICart = req.body;
  console.log('Checking out...', JSON.stringify(cart, null, 2));
  res.send({ order: '12345678' });
});

// ...
```

This endpoint doesn't do anything except log and return a fake order number. In a real world application you would interact with a database or perhaps a microservice. Since this book is about React development, we will gloss over the implementation details of this endpoint.

## Cart data-access library

Now we can add our shopping cart data-access library to the frontend application.

```
# Remember that we want a generic web library for data-access
nx g @nrwl/web:lib data-access --directory=cart
```

The cart data-access library should provide a checkout function we can use in our feature.

**libs/cart/data-access/src/lib/cart-data-access.ts**

```
import { ICart } from '@acme/shared-models';

export async function checkout(cart: ICart): Promise<{ success: boolean }> {
  const data = await fetch('/api/checkout', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(cart),
  });
  return data.json();
}
```

## Managing cart state using Redux Toolkit

The cart state will contain multiple sub-values (cart items, status flag, etc.), and we'll also need to communicate with the API endpoint. To help use manage this complexity, we can take advantage of [Redux Toolkit](#). Luckily, Nx comes with a generator to help set this up.

```
nx g redux cart --project=cart-data-access --appProject=bookstore
```

Here, we are creating a new Redux slice `cart` in the `cart-data-access` library that we created previously. As well, the generator will install the necessary npm packages for Redux Toolkit, add configure the store in `bookstore` app, and add the `cart` slice.

When the command completes, open up `apps/bookstore/src/main.tsx` and you'll see the following.

```
// ...
import { configureStore, getDefaultMiddleware } from '@reduxjs/toolkit';
import { Provider } from 'react-redux';

import { CART_FEATURE_KEY, cartReducer } from '@acme/cart/data-access';

const store = configureStore({
  reducer: { [CART_FEATURE_KEY]: cartReducer },
  // Additional middleware can be passed to this array
  middleware: [...getDefaultMiddleware()],
  devTools: process.env.NODE_ENV !== 'production',
  // Optional Redux store enhancers
  enhancers: [],
});

ReactDOM.render(
  <Provider store={store}>
    <React.StrictMode>
      <BrowserRouter>
        <App />
      </BrowserRouter>
    </React.StrictMode>
  </Provider>,
  document.getElementById('root')
);
```

Awesome, we didn't have to do anything to set up Redux in our application!

Next, let's take a look at the slice itself.

**libs/cart/data-access/src/lib/cart.slice.ts**

```
import {
  createAsyncThunk,
  createEntityAdapter,
  createSelector,
  createSlice,
  EntityState,
  PayloadAction,
} from '@reduxjs/toolkit';

export const CART_FEATURE_KEY = 'cart';

/*
 * Update these interfaces according to your requirements.
 */
export interface CartEntity {
  id: number;
}

export interface CartState extends EntityState<CartEntity> {
  loadingStatus: 'not loaded' | 'loading' | 'loaded' | 'error';
  error: string;
}

export const cartAdapter = createEntityAdapter<CartEntity>();

/**
 * Export an effect using createAsyncThunk from
 * the Redux Toolkit: https://redux-toolkit.js.org/api/createAsyncThunk
 *
 * e.g.
 * ```
 * import { useEffect } from 'react';
 * import { useDispatch } from 'react-redux';
 *
 * // ...
 *
 * const dispatch = useDispatch();
 * useEffect(() => {
 *   dispatch(fetchCart())
 * })
 * ```
 */
```



```
* }, [dispatch]);
* \``
*/
export const fetchCart = createAsyncThunk(
  'cart/fetchStatus',
  async (_, thunkAPI) => {
    /**
     * Replace this with your custom fetch call.
     * For example, `return myApi.getCarts()`;
     * Right now we just return an empty array.
     */
    return Promise.resolve([]);
  }
);

export const initialCartState: CartState = cartAdapter.getInitialState({
  loadingStatus: 'not loaded',
  error: null,
});

export const cartSlice = createSlice({
  name: CART_FEATURE_KEY,
  initialState: initialCartState,
  reducers: {
    add: cartAdapter.addOne,
    remove: cartAdapter.removeOne,
    // ...
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchCart.pending, (state: CartState) => {
        state.loadingStatus = 'loading';
      })
      .addCase(
        fetchCart.fulfilled,
        (state: CartState, action: PayloadAction<CartEntity[]>) => {
          cartAdapter.setAll(state, action.payload);
          state.loadingStatus = 'loaded';
        }
      )
  }
});
```

```
)
  .addCase(fetchCart.rejected, (state: CartState, action) => {
    state.loadingStatus = 'error';
    state.error = action.error.message;
  });
},
});

/*
 * Export reducer for store configuration.
 */
export const cartReducer = cartSlice.reducer;

/*
 * Export action creators to be dispatched. For use with the `useDispatch` ho\
ok.
 *
 * e.g.
 * ```
 * import { useEffect } from 'react';
 * import { useDispatch } from 'react-redux';
 *
 * // ...
 *
 * const dispatch = useDispatch();
 * useEffect(() => {
 *   dispatch(cartActions.add({ id: 1 }));
 * }, [dispatch]);
 * ```
 *
 * See: https://react-redux.js.org/next/api/hooks#usedispatch
 */
export const cartActions = cartSlice.actions;

/*
 * Export selectors to query state. For use with the `useSelector` hook.
 *
 * e.g.
 * ```
```

```

* import { useSelector } from 'react-redux';
*
* // ...
*
* const entities = useSelector(selectAllCart);
* \```
*
* See: https://react-redux.js.org/next/api/hooks#useselector
*/
const { selectAll, selectEntities } = cartAdapter.getSelectors();

export const getCartState = (rootState: unknown): CartState =>
  rootState[CART_FEATURE_KEY];

export const selectAllCart = createSelector(getCartState, selectAll);

export const selectCartEntities = createSelector(getCartState, selectEntities\
);

```

If you're not familiar with Redux Toolkit, you'll notice a few new utilities.

1. `createEntityAdapter` function returns a set of case reducers and selectors that makes working with normalized entities much simpler.
2. `createAsyncThunk` function returns a thunk that allows us to handle async dataflow.
3. `createSlice` function removes much of the boilerplate of Redux by allowing us to define the actions and case reducers together.
4. The case reducers—either the reducer methods, or `builder.addCase` in `extraReducers`—mutate the state rather than working with it immutably. Redux Toolkit affords the ability to perform mutable operations because it wraps the reducer around [immer](#).

The generated code is a great start for many use cases. We do need to make a few tweaks, as well as add additional selectors. So let's update the slice to the following:

**libs/cart/data-access/src/lib/cart.slice.ts**

```
import {
  createAsyncThunk,
  createEntityAdapter,
  createSelector,
  createSlice,
  EntityState,
} from '@reduxjs/toolkit';
import { ICartItem } from '@acme/shared-models';
import { checkout } from './cart-data-access';

export const CART_FEATURE_KEY = 'cart';

export interface CartState extends EntityState<ICartItem> {
  cartStatus: 'ready' | 'pending' | 'ordered' | 'error';
  error: string;
  order?: string;
}

export const cartAdapter = createEntityAdapter<ICartItem>();

export const checkoutCart = createAsyncThunk<{ order: string }, ICartItem[]>(
  'cart/checkoutStatus',
  (items) => checkout({ items })
);

export const initialCartState: CartState = cartAdapter.getInitialState({
  cartStatus: 'ready',
  error: null,
});

export const cartSlice = createSlice({
  name: CART_FEATURE_KEY,
  initialState: initialCartState,
  reducers: {
    add: cartAdapter.addOne,
    remove: cartAdapter.removeOne,
  },
  extraReducers: (builder) => {
    builder
```

```
.addCase(checkoutCart.pending, (state: CartState) => {
  state.cartStatus = 'pending';
})
.addCase(checkoutCart.fulfilled, (state: CartState, action) => {
  state.order = action.payload.order;
  state.cartStatus = 'ordered';
})
.addCase(checkoutCart.rejected, (state: CartState, action) => {
  state.cartStatus = 'error';
  state.error = action.error.message;
});
},
});

export const cartReducer = cartSlice.reducer;

export const cartActions = cartSlice.actions;

const { selectAll } = cartAdapter.getSelectors();

export const getCartState = (rootState: unknown): CartState =>
  rootState[CART_FEATURE_KEY];

export const selectCartItems = createSelector(getCartState, selectAll);

export const selecteCartStatus = createSelector(
  getCartState,
  (state) => state.cartStatus
);

export const selectOrderNumber = createSelector(
  getCartState,
  (state) => state.order
);

export const selectTotal = createSelector(selectCartItems, (items) =>
  items.reduce((total, item) => total + item.cost, 0)
);
```

## Cart feature library

Now that data-access has been sorted out, let's go ahead and add our shopping cart feature library.

```
nx g lib feature --directory=cart --appProject=bookstore
```

Recall that `--appProject` installs the new feature as a route to the bookstore application. Nx guesses what the route path should be based on your library name. Additionally, Nx will also guess where to add the new `<Link>` in your app component.

The guesses made by Nx may not be correct, so let's make sure we have the proper setup.

### apps/bookstore/src/app/app.tsx

```
// ...  
  
export const App = () => {  
  return (  
    <>  
      <GlobalStyles />  
      <Header>  
        <h1>Bookstore</h1>  
        <NavigationList>  
          <NavigationItem>  
            <Link to="/books">Books</Link>  
          </NavigationItem>  
          <NavigationItem>  
            <Link to="/cart">Cart</Link>  
          </NavigationItem>  
        </NavigationList>  
      </Header>  
      <Main>  
        <Route path="/books" component={BooksFeature} />  
        <Route path="/cart" component={CartFeature} />  
      </Main>  
    </>  
  )  
}
```

```

        <Route exact path="/" render={() => <Redirect to="/books" />} />
      </Main>
    </>
  );
};

// ...

```

Next up, let's implement our `CartFeature` component. We'll keep the implementation simple by providing the following:

1. Display each added cart item.
2. Provide a button to remove an item.
3. Show the total cost of all items.
4. Provide a button to checkout (i.e. call the checkout API).
5. Display a success message when the API returns successfully.

Something like this should do.

**libs/cart/feature/src/lib/cart-feature.tsx**

```

import styled from 'styled-components';
import { Button } from '@acme/ui';
import { useDispatch, useSelector } from 'react-redux';
import {
  cartActions,
  selectCartItems,
  selectCartStatus,
  selectOrderNumber,
  selectTotal,
  checkoutCart,
} from '@acme/cart/data-access';

const StyledCartFeature = styled.div`
  .item {

```

```

    display: flex;
    align-items: center;
    padding-bottom: 9px;
    margin-bottom: 9px;
    border-bottom: 1px #ccc solid;
  }
  .description {
    flex: 1;
  }
  .cost {
    width: 10%;
  }
  .action {
    width: 10%;
  }
};

```

```

export const CartFeature = () => {
  const dispatch = useDispatch();
  const cartItems = useSelector(selectCartItems);
  const status = useSelector(selectCartStatus);
  const order = useSelector(selectOrderNumber);
  const total = useSelector(selectTotal);
  const cartIsEmpty = cartItems.length === 0;
  return (
    <StyledCartFeature>
      <h1>My Cart</h1>
      {order ? (
        <p>
          Thank you for ordering. Your order number is <strong>#{order}</stro\
ng>
          .
        </p>
      ) : (
        <>
          {cartIsEmpty ? <p>Your cart is empty</p> : null}
          <div>
            {cartItems.map((item) => (
              <div className="item" key={item.id}>

```



```

    <span className="description">{item.description}</span>
    <span className="cost">${item.cost.toFixed(2)}</span>
    <span className="action">
      <Button onClick={() => dispatch(cartActions.remove(item.id)\
)}}>
        Remove
      </Button>
    </span>
  </div>
)}}
</div>
<p>Total: ${total.toFixed(2)}</p>
<Button
  disabled={cartIsEmpty || status !== 'ready'}
  onClick={() => dispatch(checkoutCart(cartItems))}
>
  Checkout
</Button>
</>
)}
</StyledCartFeature>
);
};

```

```
export default CartFeature;
```

Notice that we can dispatch the generated `cartActions` as well as the async `thunk` `checkoutCart` through the Redux store. This is because Redux Toolkit adds `thunk` support by default. The rest of the code is fairly standard Redux usage within a React component—select state via `useSelector` and dispatch actions using `useDispatch`.

Again, we are using `styled-components` to style the `CartFeature` component. You may choose to further extract smaller styled components out of the feature, rather than using `className` to target child elements.

## Wiring up add button in books feature

We had previously used `alert` when users clicked on the *Add* button in the books feature. Now that we have our cart feature ready, we can wire up this behavior properly.

Let's update the `BooksFeature` component as follows.

### `libs/cart/feature/src/lib/books-feature.tsx`

```
import { useEffect, useState } from 'react';
import styled from 'styled-components';
import { getBooks } from '@acme/books/data-access';
import { Books } from '@acme/books/ui';
import { IBook } from '@acme/shared-models';
import { useDispatch } from 'react-redux';
import { cartActions } from '@acme/cart/data-access';

export const BooksFeature = () => {
  const [books, setBooks] = useState<IBook[]>([]);
  const dispatch = useDispatch();

  useEffect(() => {
    getBooks().then(setBooks);
  }, []);

  return (
    <>
      <h2>Books</h2>
      <Books
        books={books}
        onAdd={(book) =>
          // Using add action from cart slice
          dispatch(
            cartActions.add({
              id: book.id,
              description: book.title,
              cost: book.price,
            })
          )
        }
      />
    </>
  );
}
```

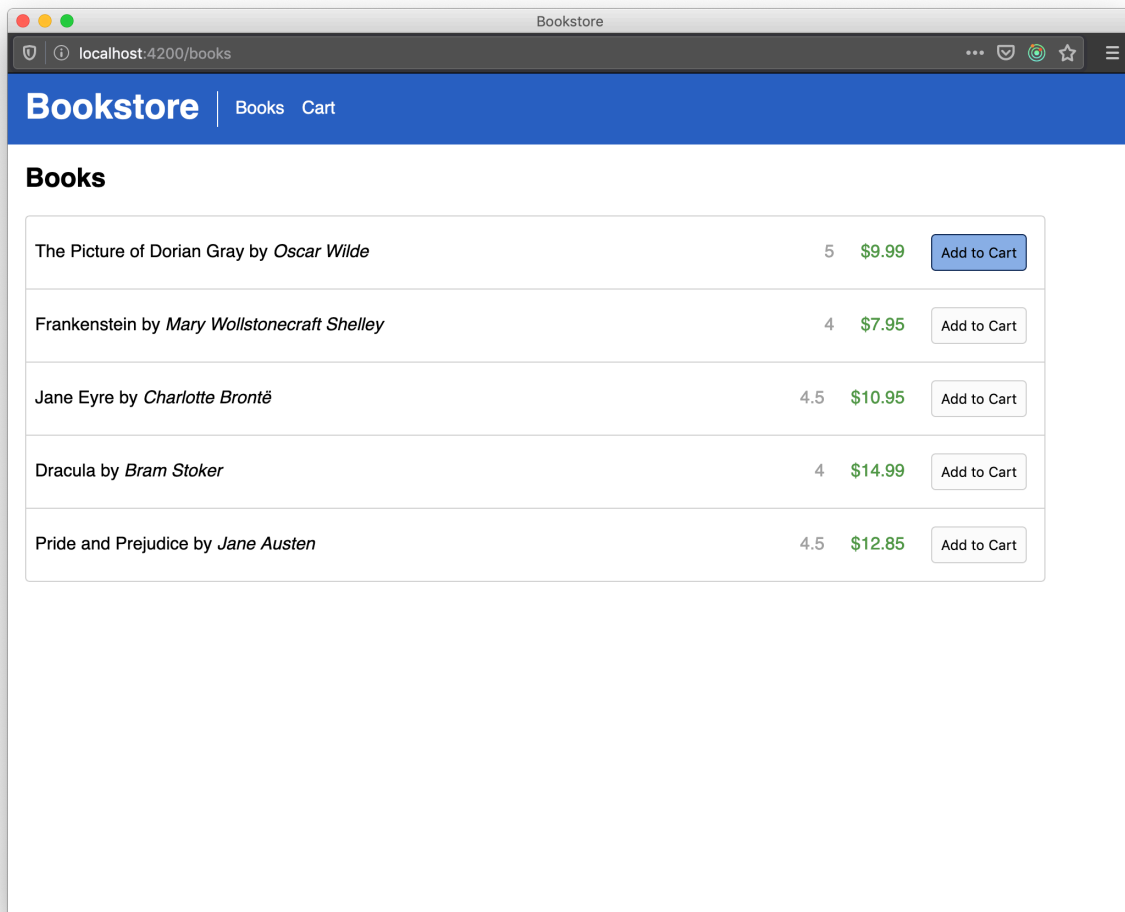
```
        })
      )
    }
  />
</>
);
};
```

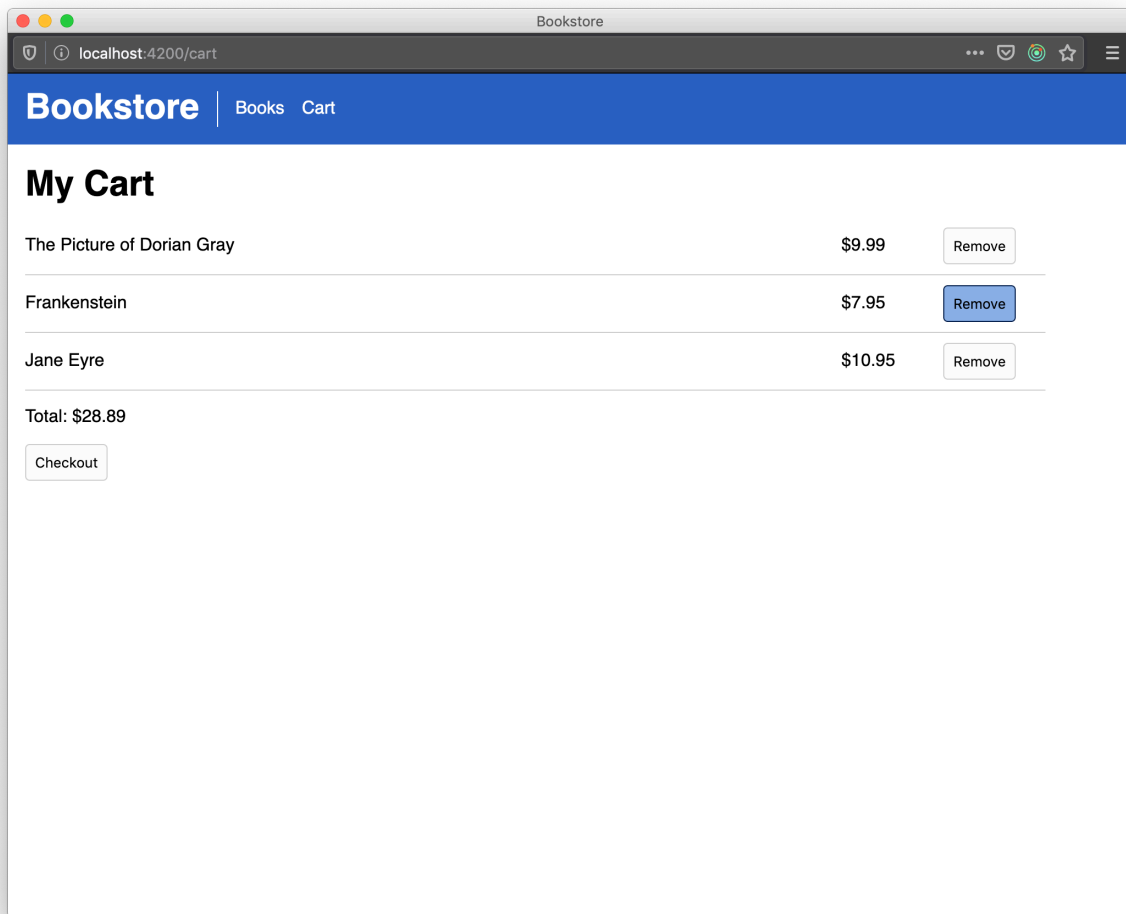
```
export default BooksFeature;
```

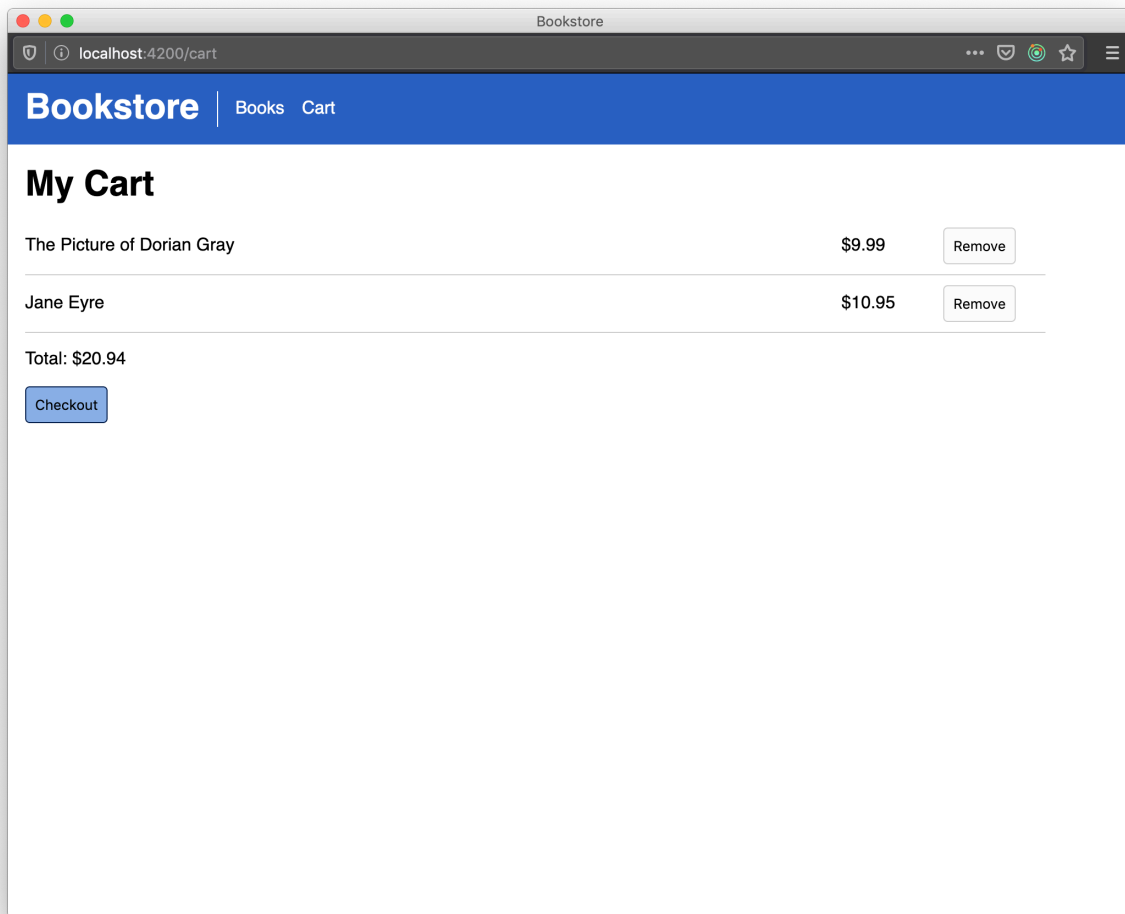
Let's look at the final result by serving up our bookstore and api apps.

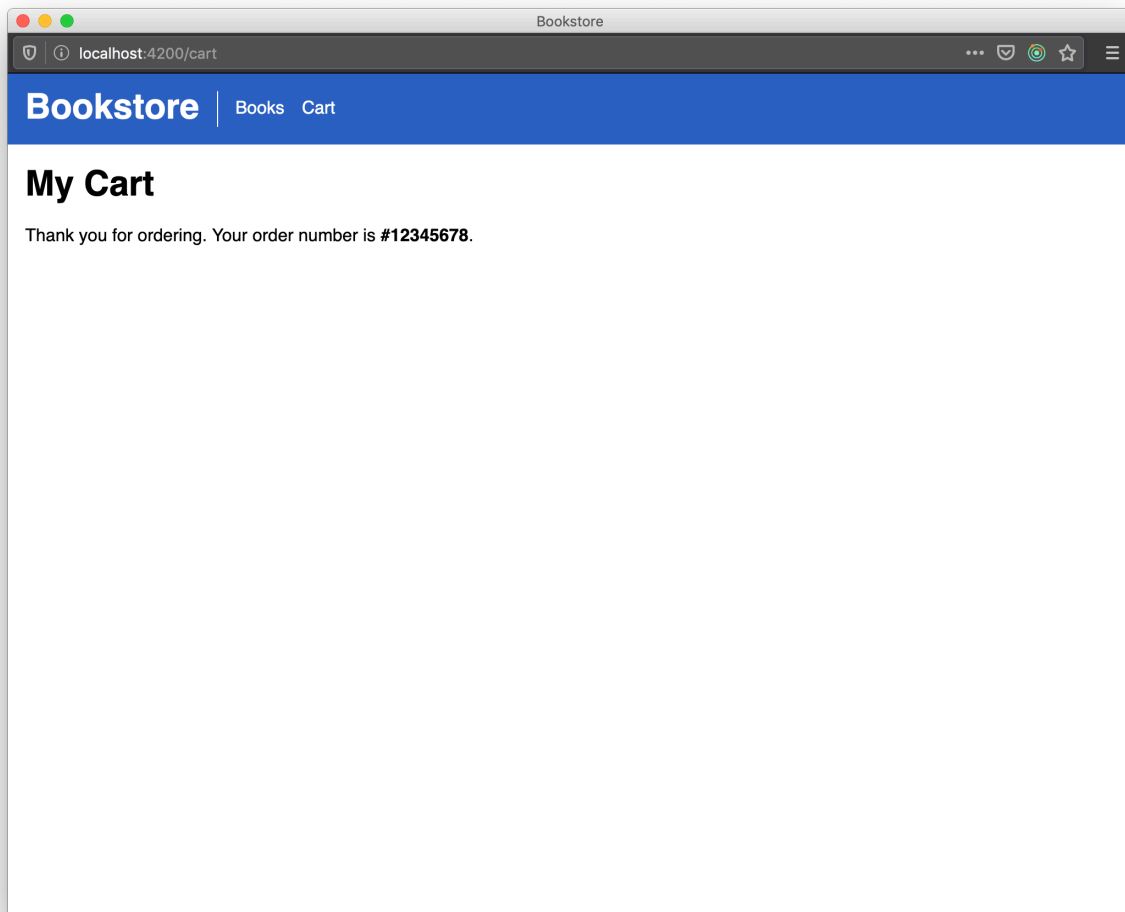
```
# Run these in different terminal windows
nx serve api
nx serve bookstore
```

Open up <http://localhost:4200> and you should be able go through the full workflow: Add books to cart, navigate to the Cart page, remove books from cart, and then check out.









Looking good! However, we're not quite yet finished. Don't forget about our tests!

```
nx affected:test
```

Tests are broken, so please fix them up. If you do get stuck though, you may refer to the solution repository: <https://github.com/nrwl/nx-react-book-example>.

You've made it this far! Now is a good time to commit our progress before looking at production builds: `git add .; git commit -m 'add cart feature'`.

## Building for production

Now that we have completed our features we can build the frontend and backend apps for running in production.

```
nx build api
nx build bookstore
```

You can also use `nx run-many --target=build --projects=api,bookstore` to build using a single command.

When both build succeed you will see the following output in the `dist` folder.

```
dist
├── apps
│   ├── api
│   │   ├── assets
│   │   ├── main.js
│   │   └── main.js.map
│   └── bookstore
│       ├── 3rdpartylicenses.txt
│       ├── assets
│       ├── favicon.ico
│       ├── index.html
│       ├── main.fc726d4f52fe3ea5.esm.js
│       ├── main.fc726d4f52fe3ea5.esm.js.LICENSE.txt
│       ├── polyfills.7e0034cfe0406d00.esm.js
│       └── runtime.bdc91b7b4b12a0bf.esm.js
```

You can run the backend application using Node, and the frontend application using any static file server solution.

e.g.



```
# Run the backend
node dist/apps/main.js

# Run the frontend
npx serve dist/apps/bookstore
```

You'll notice issues with `/api` not being available from the frontend app. There are many ways to solve this, but for the sake of simplicity we will serve the frontend app through the API server.

Update the server code with the following.

### `apps/api/src/main.ts`

```
// ...

const port = process.env.port || 3333;
const server = app.listen(port, () => {
  console.log(`Listening at http://localhost:${port}/api`);
});

// Serve built frontend app
app.use(express.static(path.join(__dirname, '../bookstore')))

// Handle browser-side routes
app.get('*', function(req, res) {
  res.sendFile('index.html', {root: path.join(__dirname, '../bookstore')});
});

server.on('error', console.error);
```

Now rebuild the API and serve.

```
nx build api
node dist/apps/api/main.js
```

Browse to `http://localhost:3333`, and you'll see the application running in production mode!

