Effective React Development with Nx

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

Jack Hsu

Effective React Development with Nx

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

Jack Hsu

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

This version was published on 2020-04-14

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.

Contents

Preface

This book is a work in progress.

First-class support for React in Nx is still new, and we will continue to improve it in order to create the best possible experience for React developers.

As Nx evolves, this book will also be updated to reflect improvements and changes to Nx. To keep up with the latest development of this book, please visit our website at https://connect.nrwl.io (TODO: point this to actual book URL).

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 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 vary 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.

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.

Why Nx?

Nx is a set of dev tools designed specifically to help teams work with monorepos. It provides an opinionated organizational structure, and a set of generation, linting, and testing tools.

Is this book for you?

Because Nx creates TypeScript source code, it helps to know some of the basics such as type annotations, and interfaces.

```
// Here's a number variable
let x: number;

// Here's an interface
interface Foo {
  bar: string;
}

// Using the interface
const y: Foo = {
  bar: 'Hello'
};
```

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.

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.

Consequently, this book might be for you if:

- 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.

Common concerns regarding monorepos

There are a few common concerns that people may have when they consider using a monorepo.

- Continuous integration (CI) is slow
- "Everyone can change *my* code"
- Teams losing their autonomy

All three of these issues will be addressed throughout this book.

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 `shopping-cart` feature, where users can add books to their cart for checkout.

Finally, this book assumes that you are using `yarn` as your package manager. If you are using `npm` then Appendix B shows you how to run the same commands using `npm`.

Chapter 1: Getting started

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

Workspace
: A folder created using Nx that contain 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 UI library.

Now, let's create our workspace.

Creating an Nx workspace

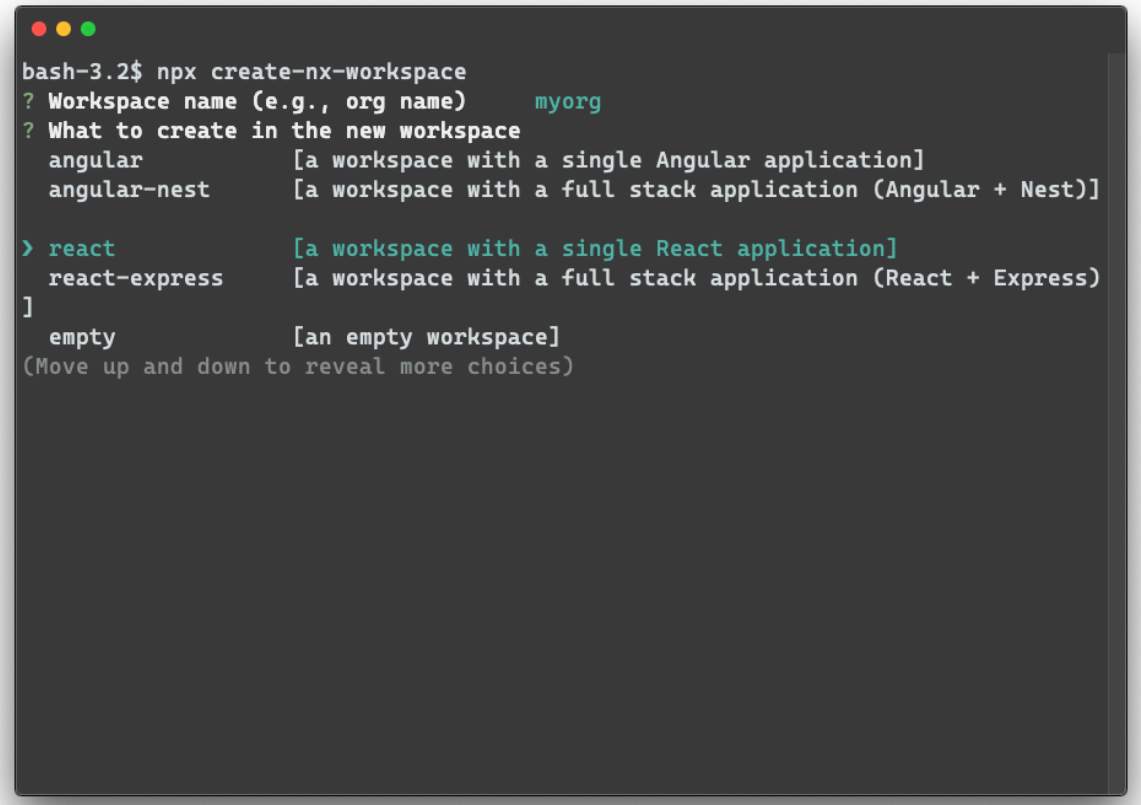
You can create the workspace as follows:

```
npx create-nx-workspace
```

Note: The `npx` binary comes bundled with NodeJS. It allows you to conveniently install then run a Node binary.

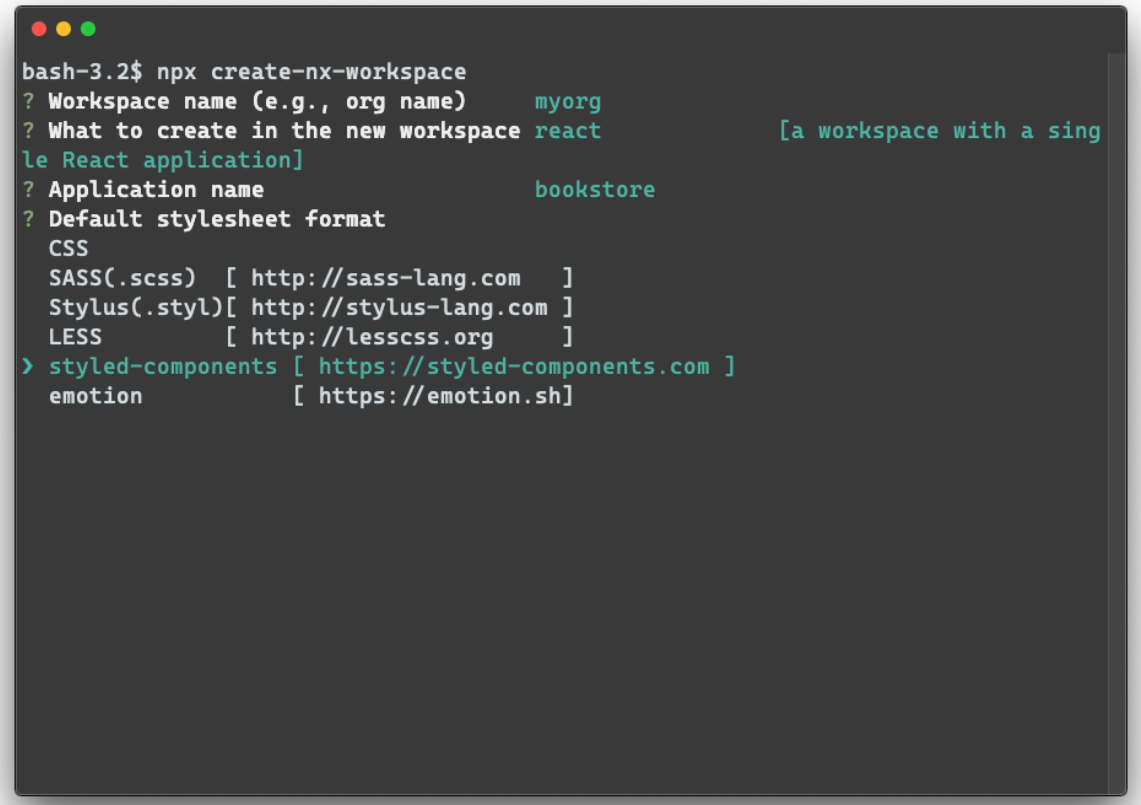
Nx will ask you for a **workspace name**. Let's use `myorg` as it is the name of our imaginary organization. The workspace name is used by Nx to scope our libraries, just like npm scoped packages.

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



Creating a workspace

Lastly, 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.



Choosing a style option

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

```
myorg
├── apps
│   ├── bookstore
│   │   ├── src
│   │   │   ├── app
│   │   │   ├── assets
│   │   │   ├── environments
│   │   │   ├── favicon.ico
│   │   │   ├── index.html
│   │   │   ├── main.tsx
│   │   │   └── polyfills.ts
│   │   ├── browserslist
```

```
│   │   ├── jest.config.js
│   │   ├── tsconfig.app.json
│   │   ├── tsconfig.json
│   │   └── tsconfig.spec.json
│   └── bookstore-e2e
├── libs
├── tools
│   ├── schematics
│   └── tsconfig.tools.json
├── README.md
├── nx.json
├── package.json
├── tools
├── tsconfig.json
└── workspace.json
```

The `apps` folder contain 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 `bookstore` application.

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

The `tools` folder can be used for scripts that are specific to the workspace. The generated `tools/schematics` folder is for Nx's workspace schematics feature which we cover in Appendix A.

The `nx.json` file configures Nx (as we'll see in Chapter 4).

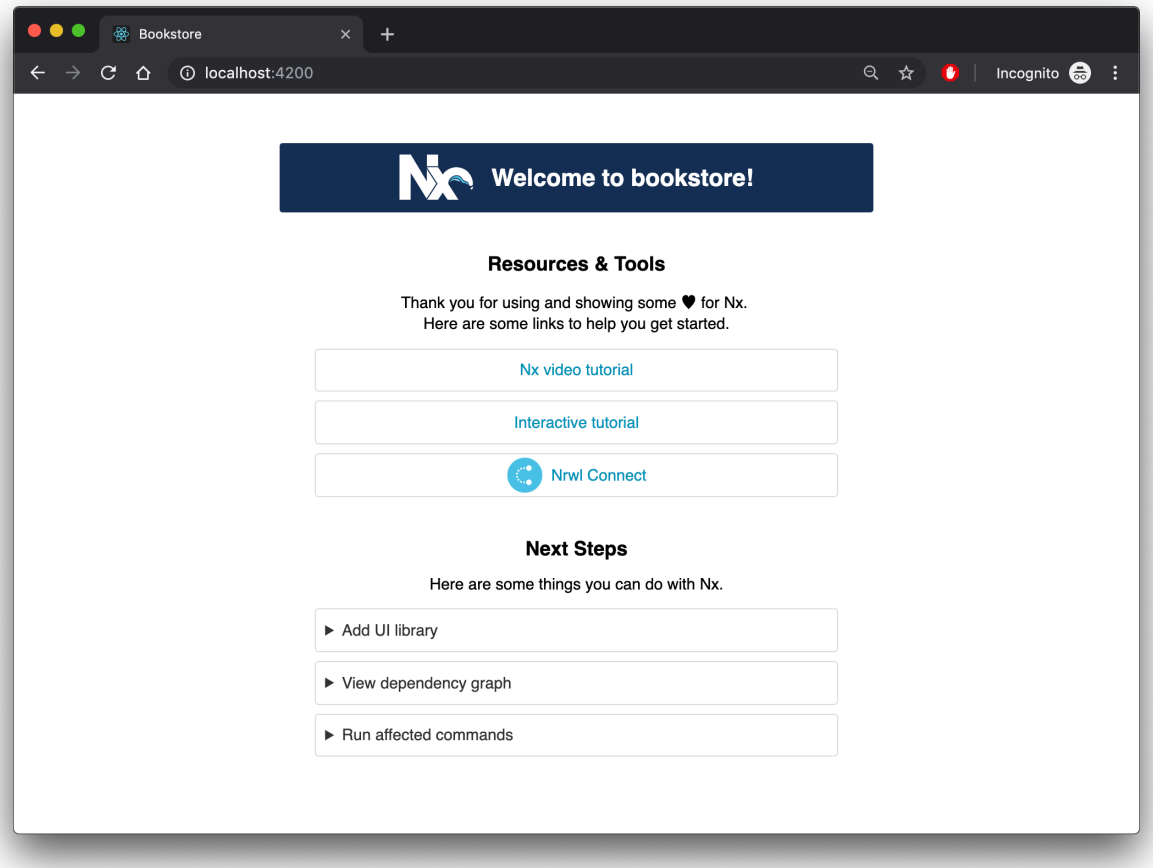
The `workspace.json` file configures our projects (applications and libraries) within the workspace. Here, you can specify what and how commands like `lint`, `test`, and `e2e` are run.

To serve the application, use this command:

```
nx serve bookstore
```

The above command will build the `bookstore` application, then start 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

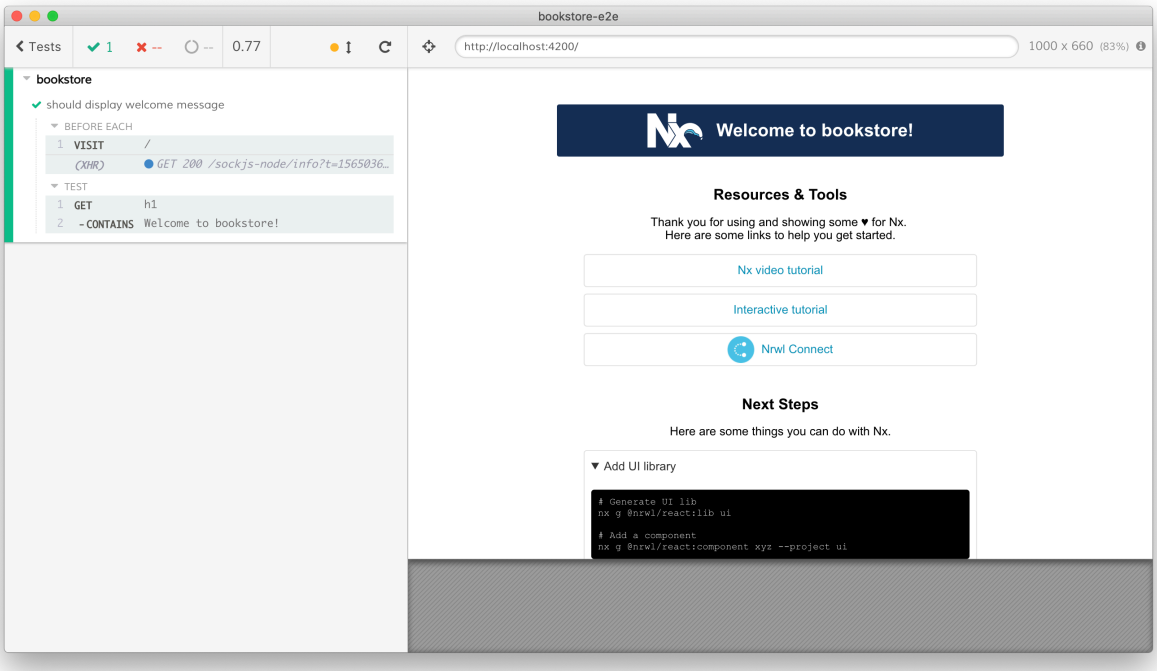
Nx commands

Nx comes with a set of targets that can be executed on our projects. You run a target by running commands in the form: `nx [target] [project]`.

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

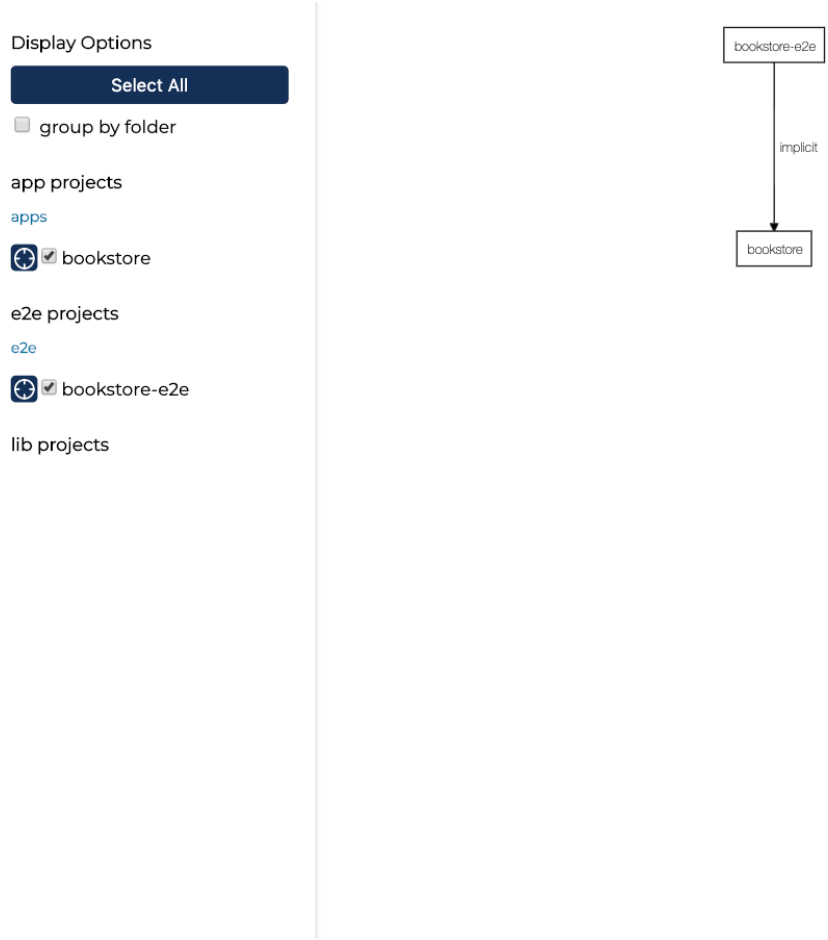
```
# Run a linter for the application
npx nx lint bookstore

# 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!



`nx e2e bookstore-e2e`

Lastly, Nx allows us to examine the dependency graph of our workspace with the `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 the install has worked by issuing the command `nx --version`.

Now you will be able to run Nx commands without going through `yarn` (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 `yarn`.

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 React from 'react';
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 React from 'react';
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 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

An 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.

Types of libraries

In a workspace, libraries are generally 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 the next 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 schematics 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
```

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 not needed, but it is useful because Nx will do three things for us. It is aliased to `-a`.

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.

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

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

import { BooksFeature } from '@myorg/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 React from 'react';
import ReactDOM from 'react-dom';
import App from './app/app';

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

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

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

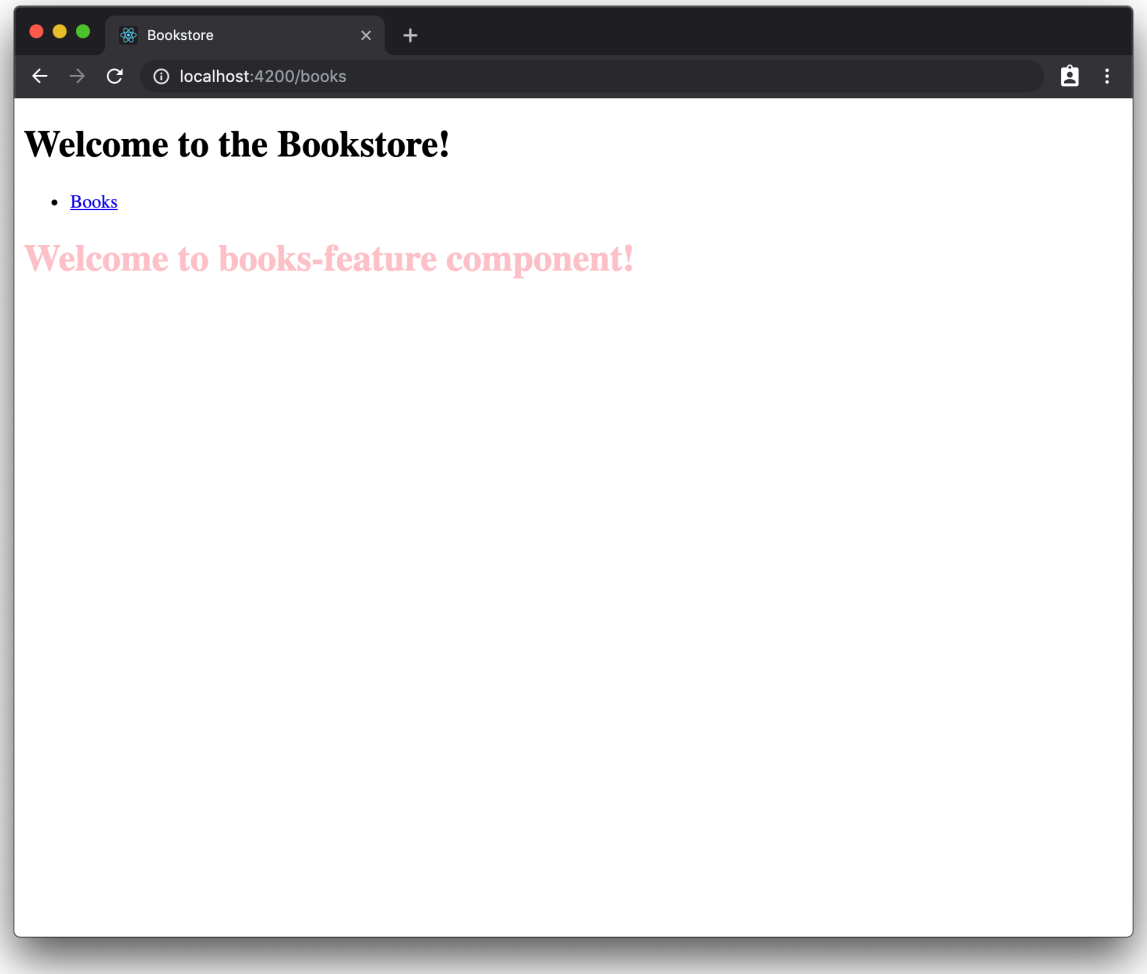
Be aware that when you add a new project to the workspace, you must restart your development server. This restart is necessary in order for the TypeScript compiler to pick up new library paths, such as `@myorg/books/feature`.

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 --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 our their documentation.

Back to the example. You should have a new folder: `libs/ui`.

```
myorg
├── (...)
├── 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
nx g component Button --project ui --export
nx g component Header --project ui --export
nx g component Main --project ui --export
nx g component NavigationList --project ui --export
nx g component NavigationItem --project ui --export
```

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 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.

This component is useful for overriding global style rules such as `body { margin: 0 }`.

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

```
import React from 'react';
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 React, { 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 React, { 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 React, { 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 React, { 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 React, { 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;
```

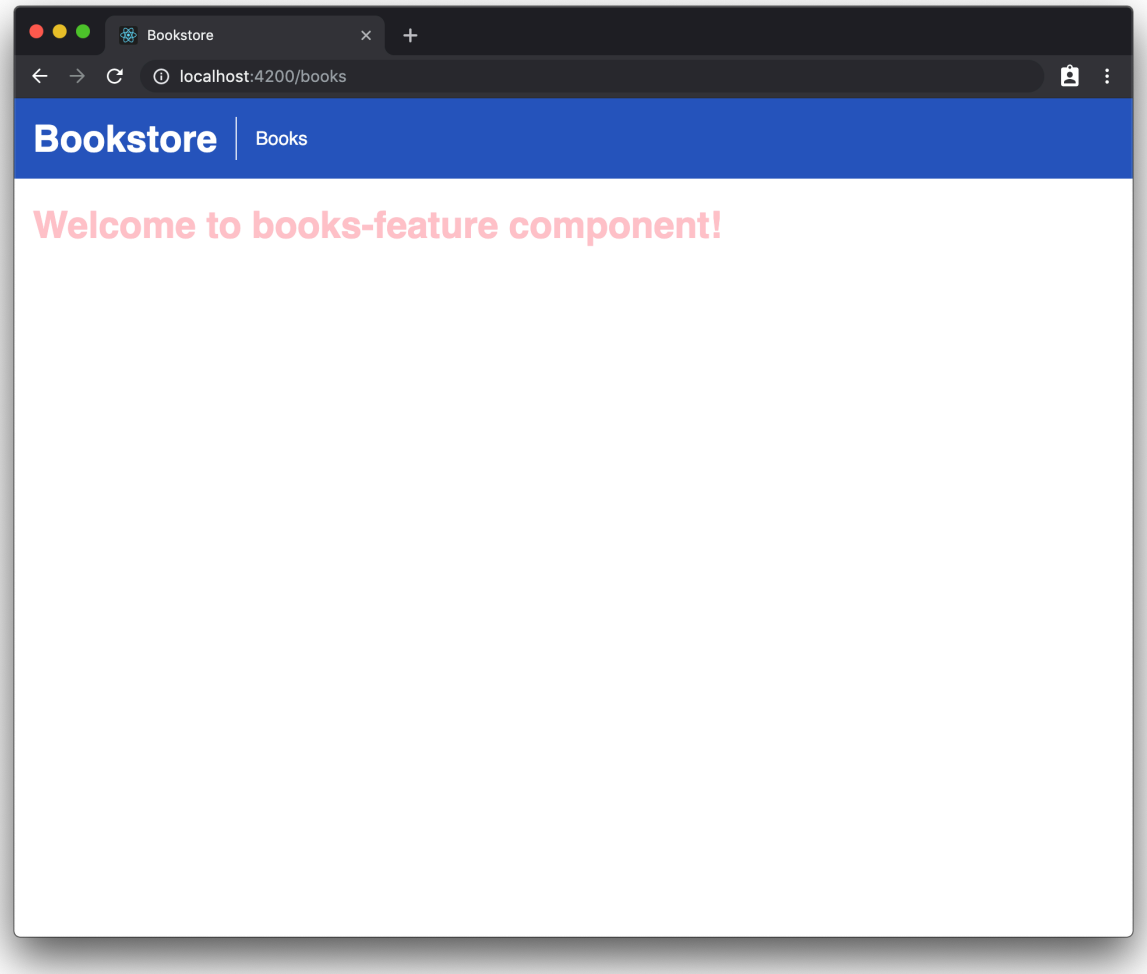
Using the UI library

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

apps/bookstore/src/app/app.tsx

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

import { BooksFeature } from '@myorg/books/feature';
import {
  GlobalStyles,
  Header,
  Main,
  NavigationItem,
  NavigationList
} from '@myorg/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 libraries'
```

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
```

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`) schematic provided by the `@nrwl/web` collection.

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

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

In this case, the `@nrwl/web:lib` schematic 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.

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

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

  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 effects from bleeding into them, thus making them easier to understand and test.

Again, we will see in Chapter 3 how Nx enforces module boundaries.

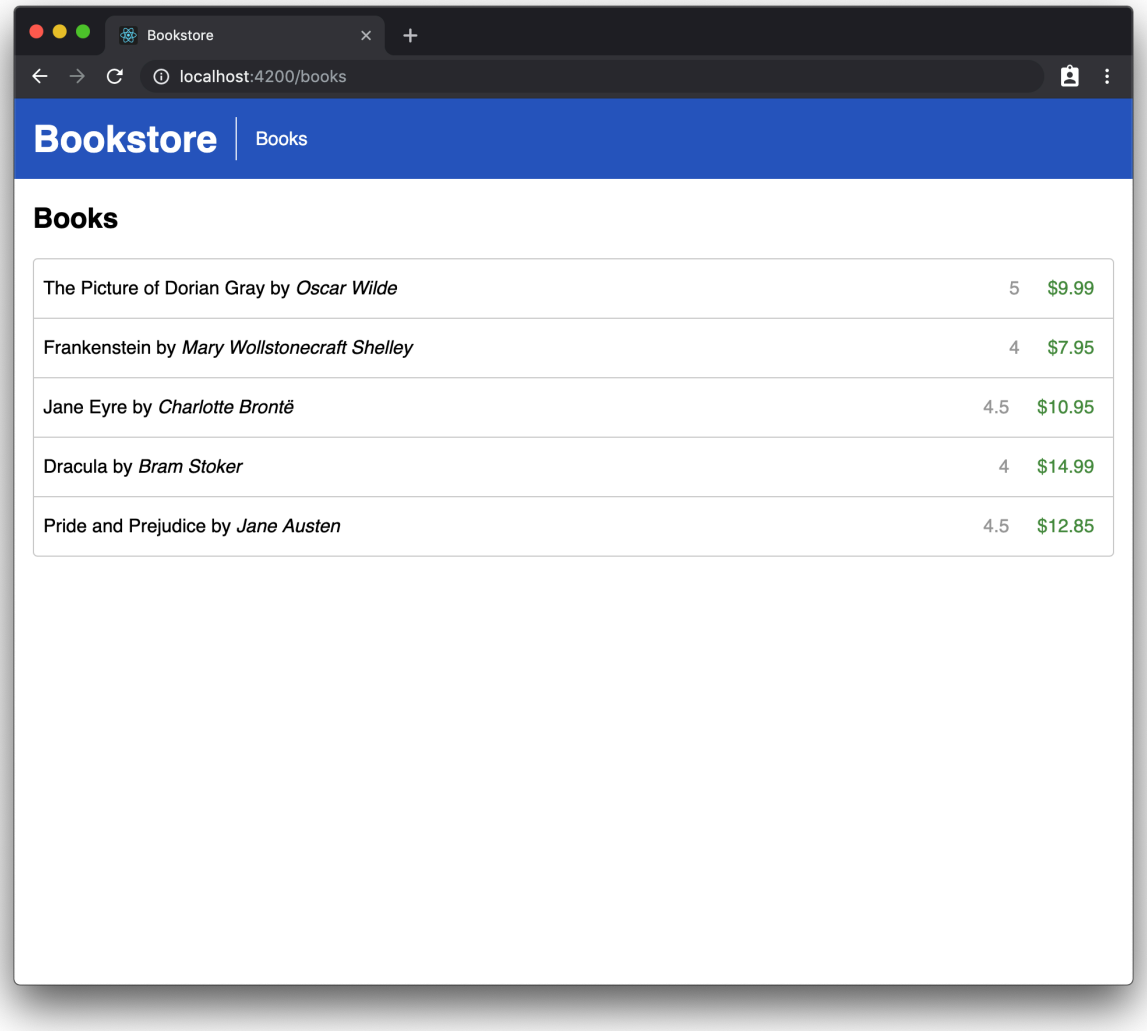
libs/books/ui/src/lib/books/books.tsx

```
import React from 'react';
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 React from 'react';
import styled from 'styled-components';
import { Button } from '@myorg/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.

Key points

Libraries are separated into four types: *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 schematic from that specific collection rather than taking the workspace's `defaultCollection`.

Chapter 3: Working effectively in a monorepo

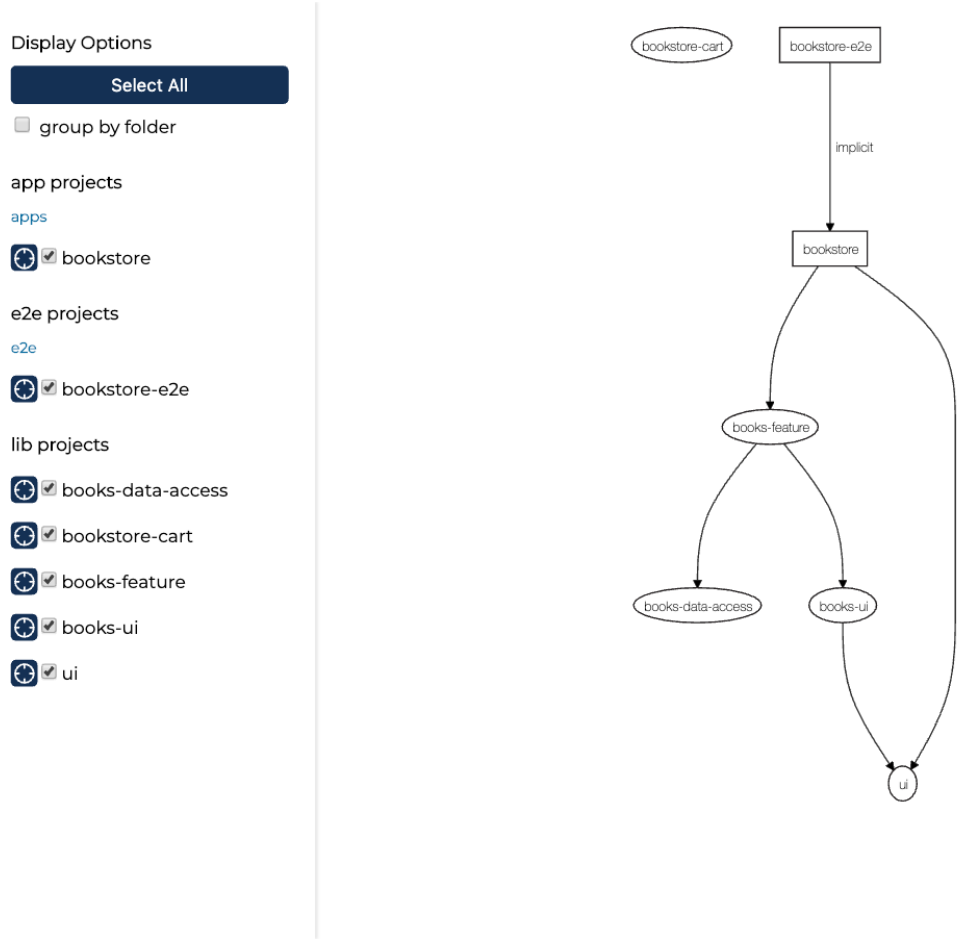
In the previous two chapters we set up a `bookstore` application that renders a list of books for users to purchase.

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.

Understanding and verifying changes

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 React from 'react';
import styled from 'styled-components';
import { Button } from '@myorg/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) => {
  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={() => onAdd(book)}>Add to Cart</Button>
      </span>
    </StyledBook>
  );
};

export default Book;
```

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

```
import React from 'react';
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>
  );
};
```

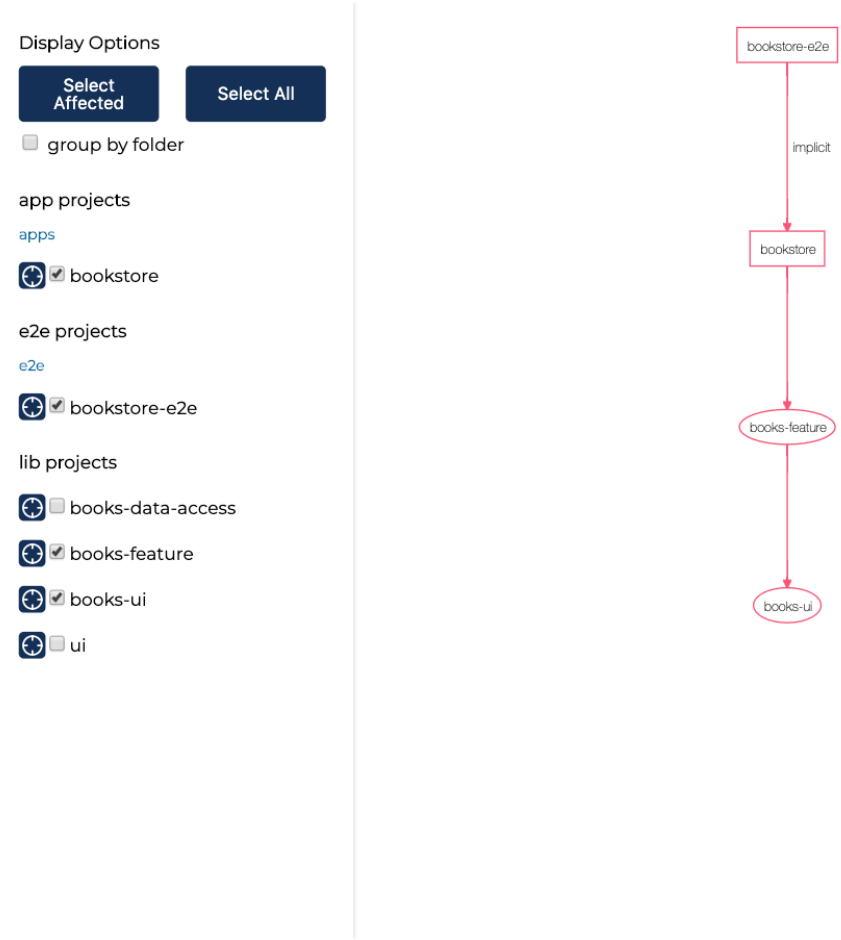
```
export default Books;
```

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

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

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

  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;
```

We can ask Nx to show us how this change *affects* the projects within our workspace.

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



Affected dependencies

As we can see, Nx knows that the `books-ui` library has changed from the `master` branch; it has indicates the dependent projects affected by this change in *red*. Furthermore, Nx also allows us to retest only the affected projects.

We can **lint** the projects affected by the changeset.

```
nx affected:lint --parallel
```

Or run **unit tests** for the affected projects.

```
nx affected:test --parallel
```

Or even run **e2e tests** for the affected projects.

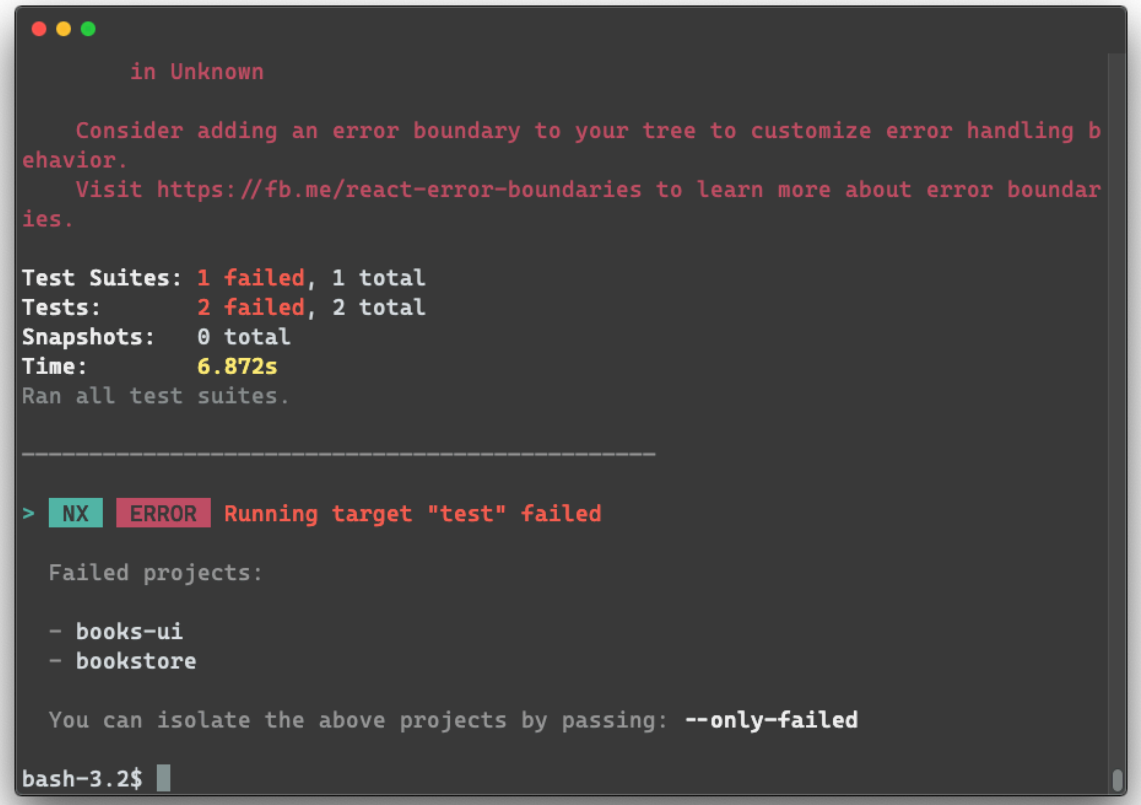
```
nx affected:e2e
```

Nx topologically sorts the projects so they are run from bottom to top. That is, projects at the bottom of the dependency chain are run first. We're also using the `--parallel` option to enable Nx to run our projects in parallel.

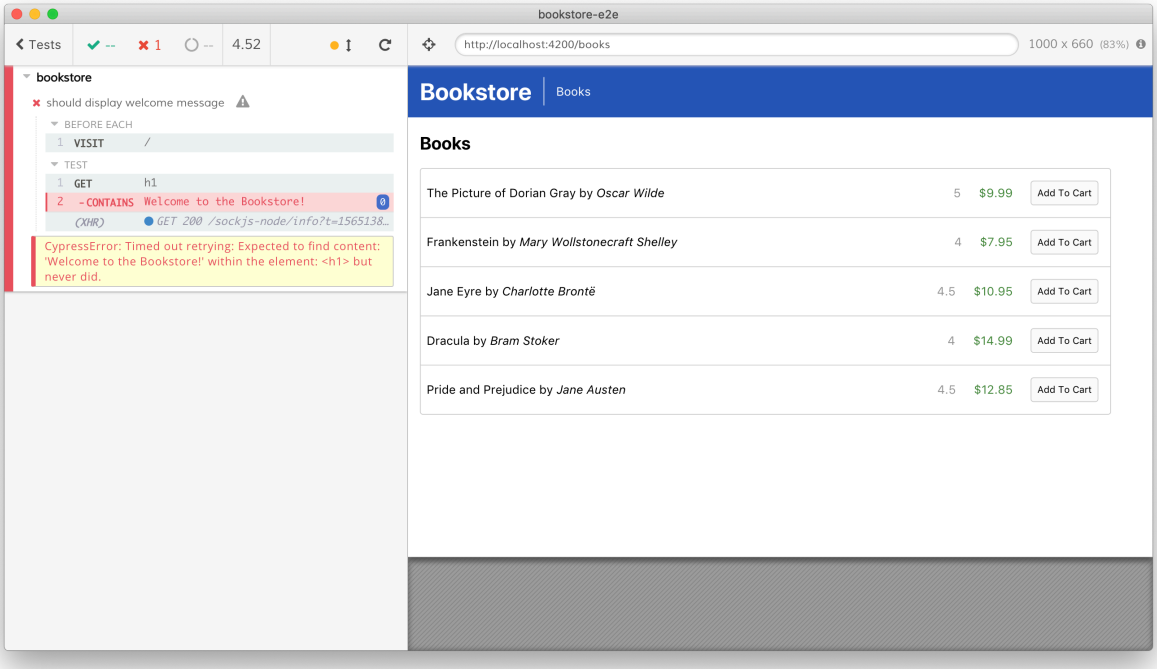
And just as with the `affected:dep-graph` command, the default base is the `master` branch. This default branch should be correct most of the time, but can be changed with the `--base=[branch]` option. For example, if your team uses git-flow then you may want to use `--base=develop` when doing work on the development branch.

> 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 runners by specifying them in `workspace.json`. See Appendix A for more information.

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



`nx affected:test` failed



`nx affected:e2e` failed

I'll leave it to you as an exercise to fix the broken unit and e2e tests. **Tip:** Run the tests in watch mode by passing the `--watch` option so that tests are rerun whenever the source or test code change.

There are three additional affected commands in Nx.

1. `nx affected:build` – Builds only the affected apps. We'll go over build and deployment in Chapter 5.
2. `nx affected:apps` – Lists out all applications affected by the changeset.
3. `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.

Adding the API application

By the way, now is a good time to commit your changes if you haven't done so already.

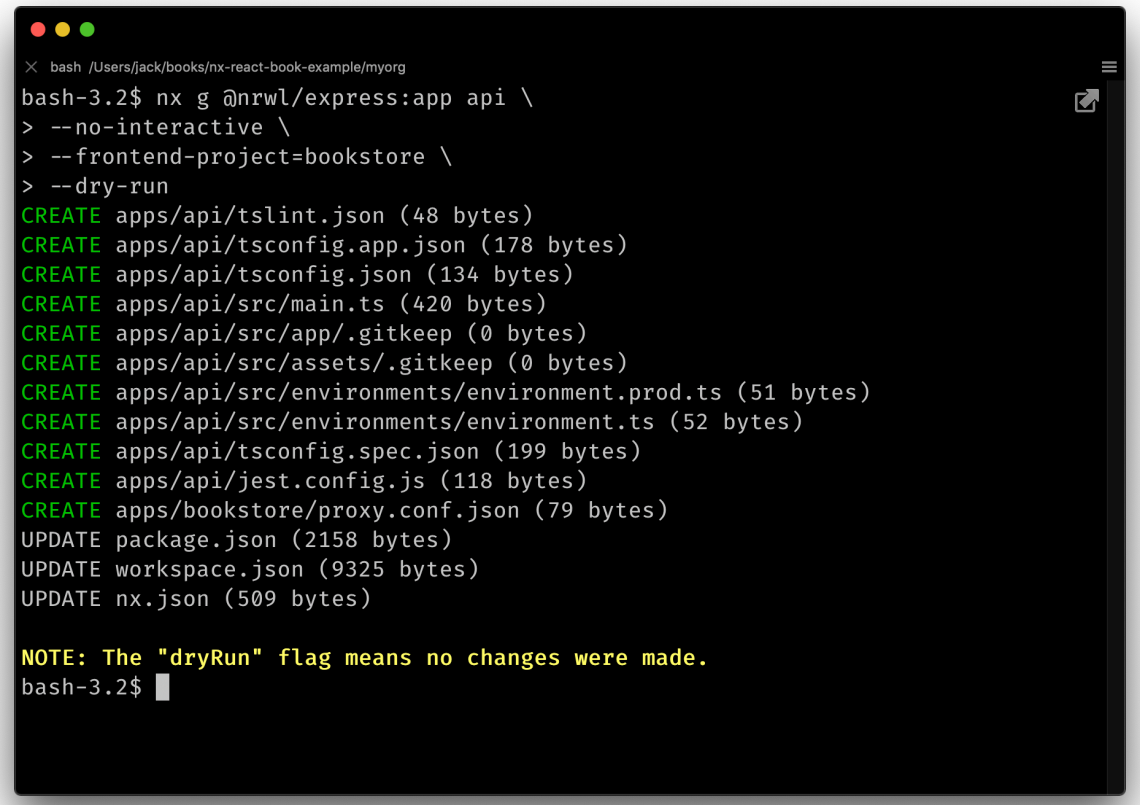
So far our `bookstore` application does not communicate with a real backend service. Let's create one using the Express framework.

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

```
yarn add --dev @nrwl/express
```

Then we can do a dry run of the generate command.

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



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 proxy configuration to the `bookstore` application such that requests going to `/api/*` will be forwarded to the API

And 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 nice 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
    },
    {
```

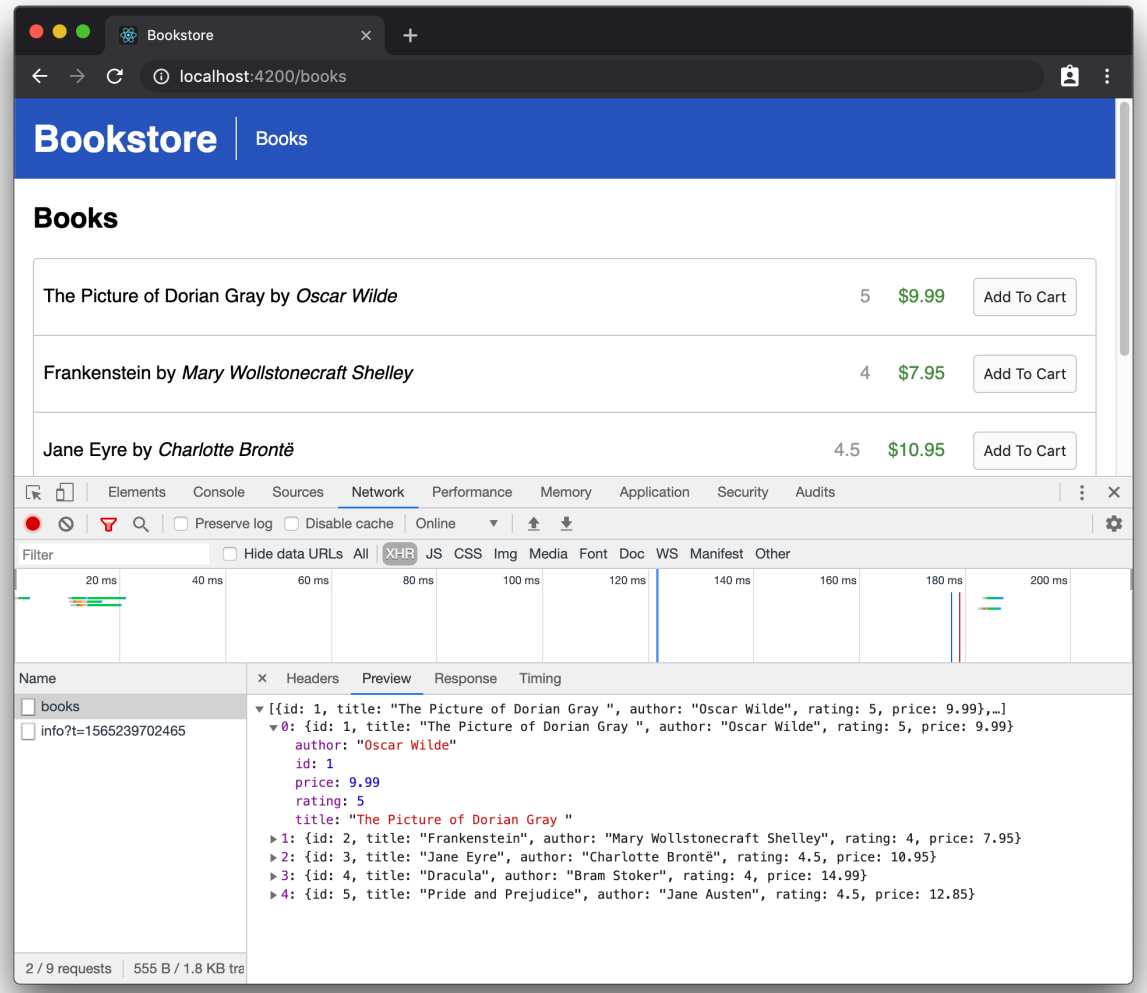
```
      id: 4,
      title: 'Dracula',
      author: 'Bram Stoker',
      rating: 4,
      price: 14.99
    },
    {
      id: 5,
      title: 'Pride and Prejudice',
      author: 'Jane Austen',
      rating: 4.5,
      price: 12.85
    }
  ];
  res.send(books);
});

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

Finally, let's update our data-access library to call the proxied 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`) 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.



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 '@myorg/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 '@myorg/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

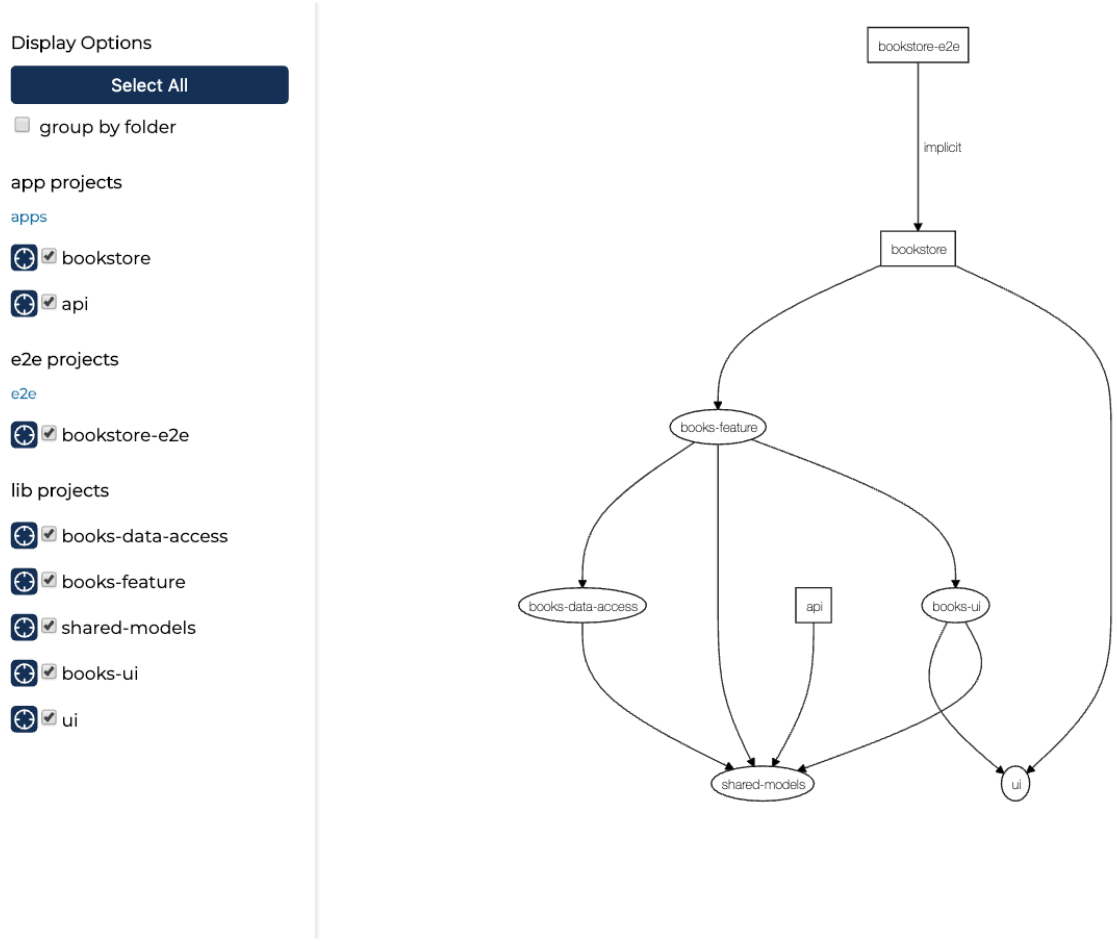
```
...
import { IBook } from '@myorg/shared-models';

export const BooksFeature = () => {
  // Replace any with IBook
  const [books, setBooks] = useState([] as 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 '@myorg/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 '@myorg/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*. This means that the corresponding pull-request contains 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 time *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 shouldn't.

Prettier was created to stop these endless debates over code style. It is highly opinionated and provides minimal configuration options. And 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 comes prepackaged with Prettier. 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
```

Lastly, you may want to set up a pre-commit git hook to run `nx format:write` so we can ensure 100% conformance whenever code is checked in. For more details please refer to Appendix C.

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

(TODO: Add content)

Appendix A: Shallow dive into workspace.json

(TODO: Add content)

Appendix B: Using `npm` instead of `yarn`

We make light use of `yarn` so most of this book should remain the same if you follow along using `npm`.

There are four places where you will need to run different commands.

1. `yarn create ...` becomes `npm init ...`.
2. `yarn add ...` becomes `npm install --save ...`.
3. `yarn global add ...` becomes `npm install -g ...`.
4. `nx ... becomes` npm run nx ...‘.

In Chapter 1 when you create the workspace, you should run `npm init nx-workspace`.

The examples in this book assume you have `nx` installed globally. So go ahead and run `npm install -g @nrwl/cli`.

Appendix C: Pre-commit git hook to automatically format code

(TODO: Add content)