Developing Client-Side Dynamic Web Applications

Exercise Manual

ROITraining
Maximize Your Training Investment

Fidelity LEAP
Technology Immersion Program

This page intentionally left blank.

Table of Contents

This page intentionally left blank.

General Instructions

Exercises are done by an individual student or in pairs. Workshops are done by assigned workgroups sharing files electronically. In lab breakout groups, it is best practice for at least one student to share their screen so students can support each other in hands-on lab work.

This page intentionally left blank.

Chapter 1: Introduction to Visual Studio Code

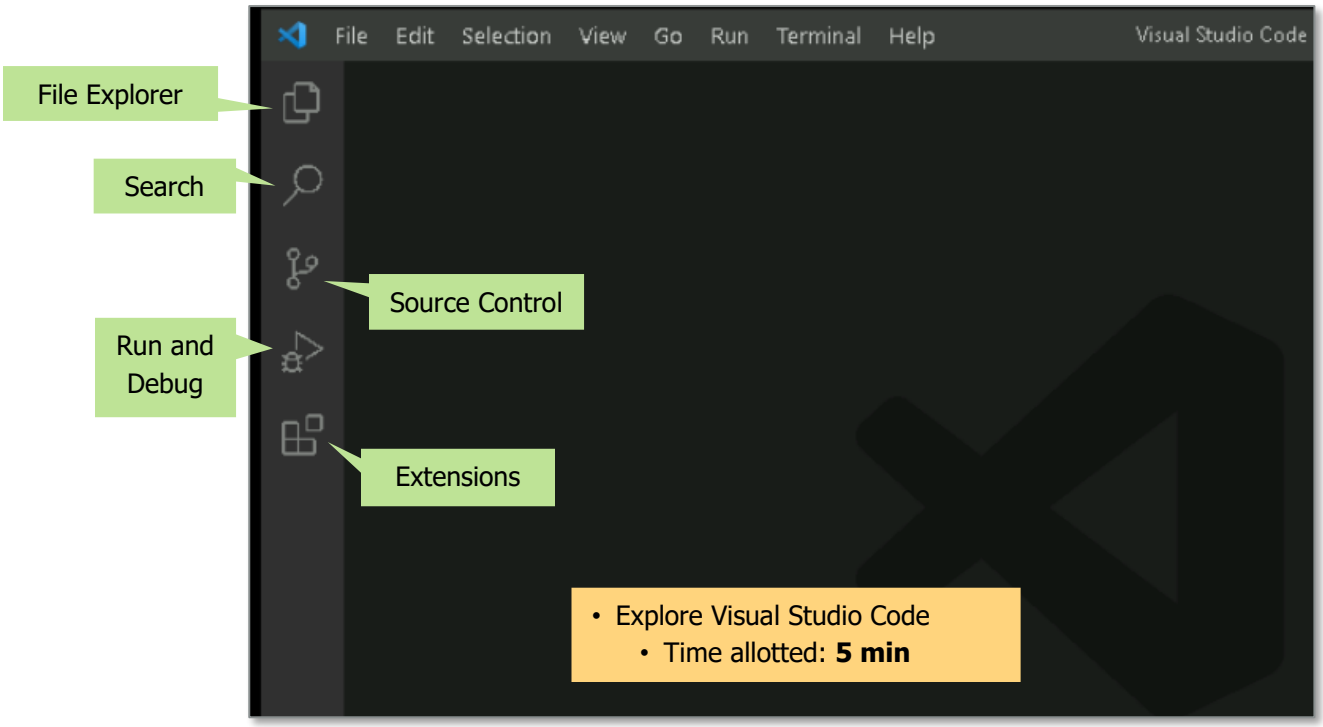
Exercise 1.1: Introduction to Visual Studio Code

Time: 20 minutes
Format: Instructor-led exercise

The instructor will demonstrate several features of Visual Studio Code. Associates will perform the same steps to verify that they can successfully use VSC to create and execute a simple web page.

Getting to Know Visual Studio Code (VSC)

1. Start Visual Studio Code. The Welcome page should be displayed. Your instructor will point out several features of the VSC environment.

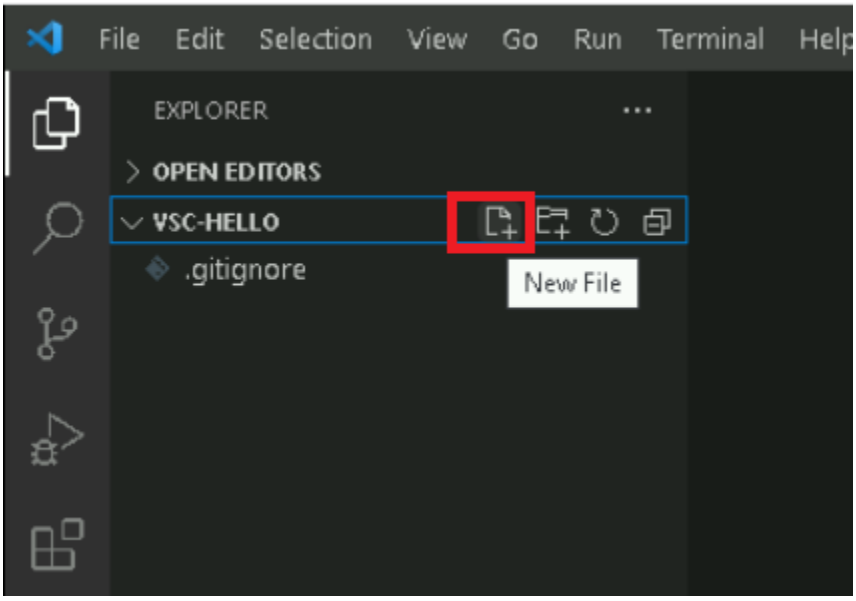


2. Click the **Extensions** icon.
    a. In the search field, type: `html boilerplate`.
    b. Select **HTML Boilerplate** from the list.
    c. Click **Install**.
3. Do the same with:
    a. `Angular Language Service`
        - Provides a rich editing experience for Angular templates.
    b. `open in browser` by TechER
        - A quick way of opening the current file in your default browser.
4. Review the following additional extensions, which are already installed:
    a. Git History
    b. Compare Folders
    c. Partial Diff
    d. ToDo Tree

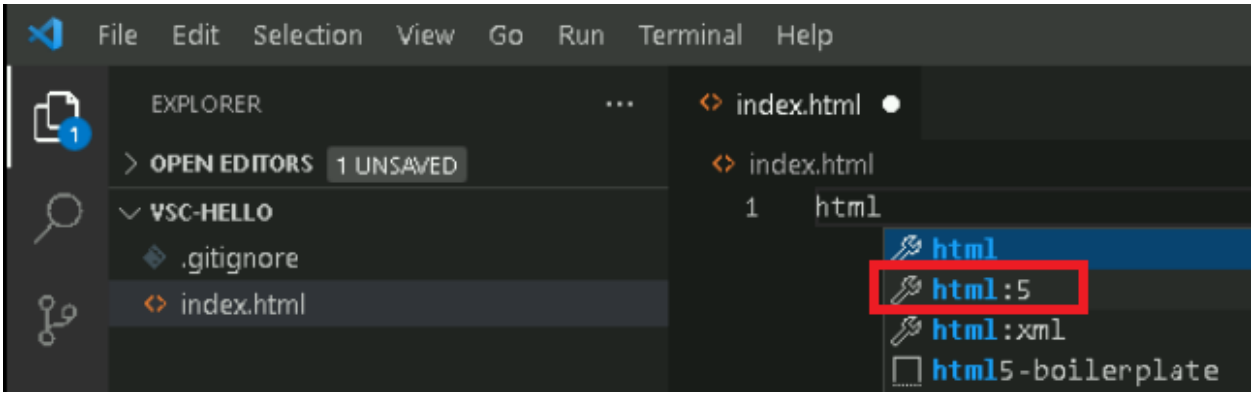
Generate Your First Web Page with VSC

5. Copy the exercise files to your virtual machine.
    a. Copy `M:\FSE\WebApps\WebAppsClient.zip` to your `D:` drive.
    b. Right-click the zip file and select **7-Zip > Extract Here**. (Don't extract directly from `M:` because it causes problems when too many people access the drive at the same time.)

6. In VS Code, select **File > Open Folder** and choose folder `WebAppsClient\Ch01\VSC-Hello`.
    a. After the folder is open, hover the mouse cursor over the `VSC-Hello` bar and click the **New File** icon.
        - Name it: `index.html`

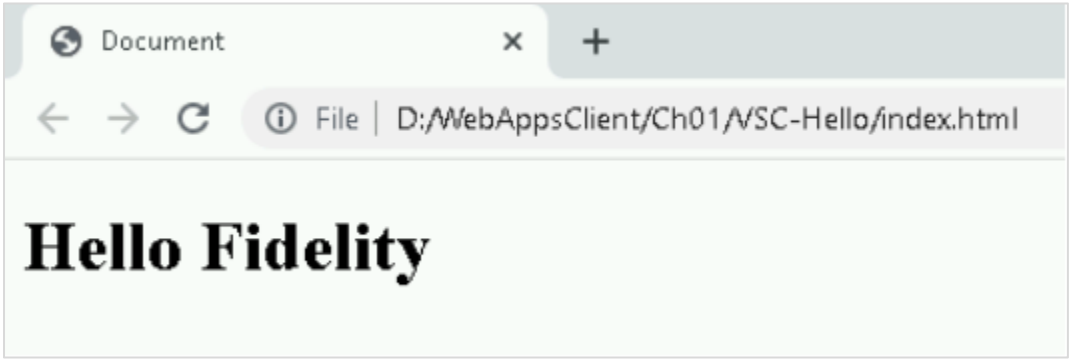


7. Switch to the `index.html` tab, type `html`, and click **html:5 ->**
    a. Boilerplate code is implemented to create a basic page structure.



Executing Web Page

8. First, add a heading to the web page:
    a. Add `<h1>Hello Fidelity</h1>` between the `<body>` tags.
    b. Save file. Or, set Visual Studio Code to **Auto Save**.
    c. Right-click the VS Code Explorer file and choose **Open in Default Browser**.
        - Set the Default Browser to Google Chrome if given the option.
        - You should see something like the following:



Chapter 2: Standardizing Presentation with HTML and CSS

Exercise 2.1: Applying CSS Styling

Time: 20 minutes
Format: Individual exercise

1. Open the files `Ch02\ApplyCssStyling\index.html` and `Ch02\ApplyCssStyling\style.css`.
    a. Complete the TODO steps in both files.
    b. To see your changes, you will need to reload the web page.

This page intentionally left blank.

Chapter 3: Advanced HTML and CSS

Exercise 3.1: Applying Advanced CSS Styling Effects

Time: 20 minutes
Format: Individual exercise

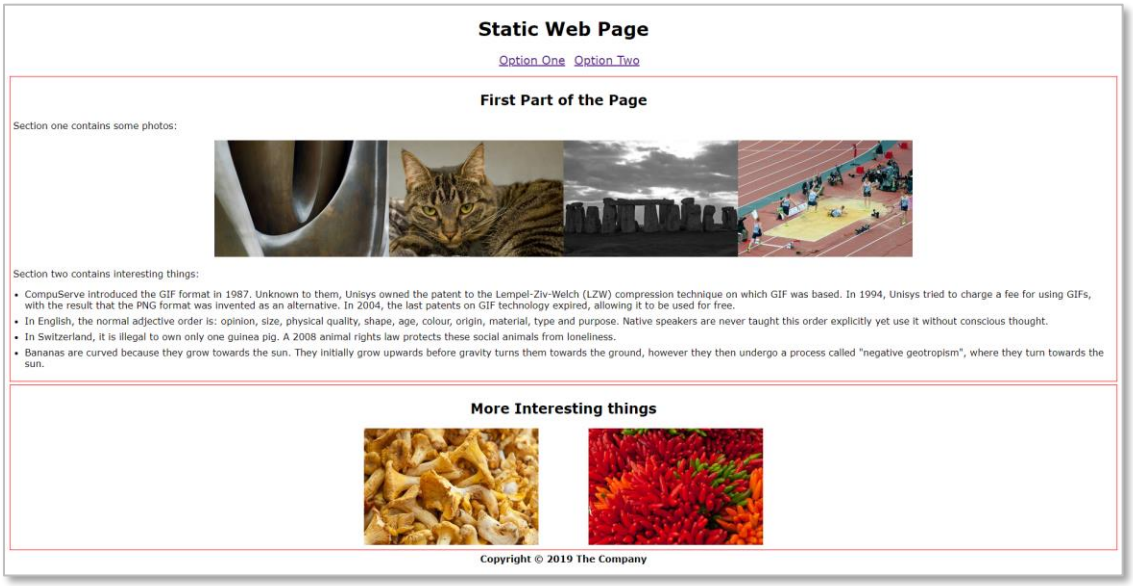
1. Continue working with your solution to the previous exercise.
2. Change the way the photographs are displayed:
    a. Make their width 300px.
    b. Add a drop-shadow.
    c. Make sure the first one is outside the normal text flow and on the left. The second one should be on the right. Make sure there is some spacing between the images and the text.
    d. Add a transition so that the images are displayed at their full width (600px) when the mouse pointer is over them.
3. Add a suitable gradient to the page background.
4. To see your changes, you will need to reload the web page.

Exercise 3.2: Static Web Pages (Optional)

This is an optional exercise that can be done if time and interest permit.

Time: 30 minutes
Format: Individual or pair exercise

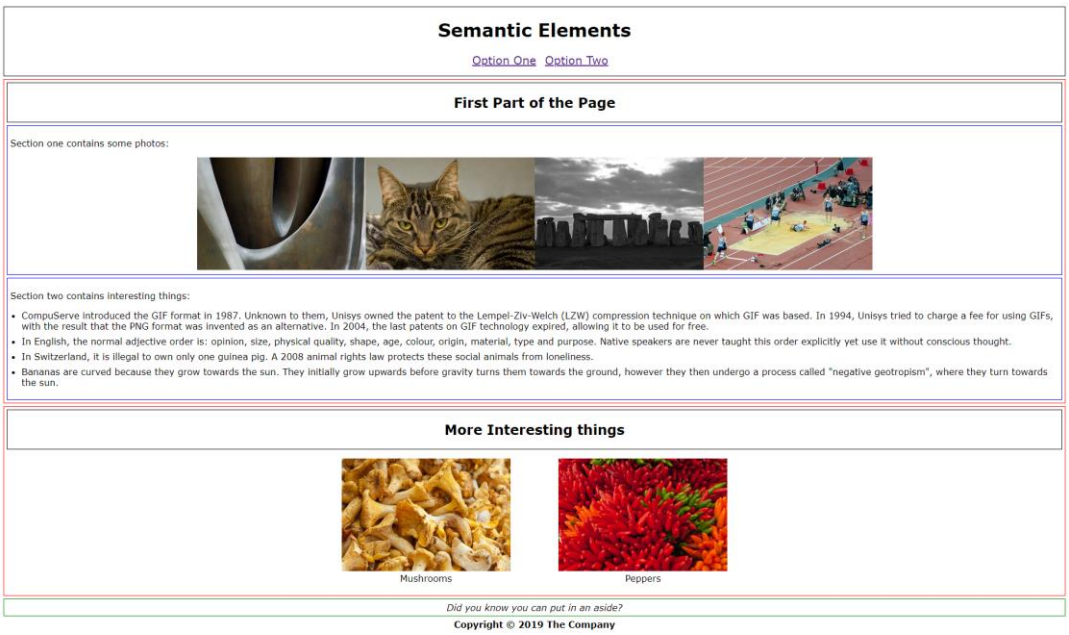
1. Open the project `Ch03\StaticWebPage` and create a static web page with a separate CSS file.
2. Your web page should look somewhat like the one below, but in particular:
    a. There should be an overall page header containing a heading and links.
    b. There should be a copyright message at the foot of the page.
    c. The rest of the page should be divided into two distinct parts with some indication of which part is which (e.g., a border).
    d. The first of the two parts should be further subdivided into a section with photos and another with bullets. Align the images seamlessly.
    e. The second of the two parts should contain other photos or text. If you use images, arrange them differently from those in the previous section.
    f. Alternatively, create additional pages for the links and/or click the pictures.
3. Please do not make changes at random: have a plan and aim to stick to it as closely as possible. If you need ideas, you can try to reproduce the sample page exactly.



Exercise 3.3: Creating an HTML5 Page Using Semantic Tags

Time: 20 minutes
Format: Individual exercise

1. The idea is to produce the web page shown or something very similar. You can either work from the exercise starter or your solution to Exercise 3.2 if you did it.
2. Use the following semantic tags to *replace* divs or other elements:
    a. `header`: to contain the page level heading and the links.
    b. `nav`: to contain the top-level links.
    c. `article`: to replace the two parts of the page.
    d. `section`: to differentiate the two sections of the first part.
    e. `header`: to contain each of the headings inside individual articles or sections.
    f. `footer`: to include the copyright message.
    g. `figure`: to contain the images in one article. In the solution, we have chosen to do this for the second section. Give each figure a `figcaption`.
    h. Add an `aside` with a suitable message.
3. Add borders to the semantic tags: black for header, red for an article, blue for section, and green for aside.



Exercise 3.4: Creating an HTML Form with Validation

Time: 20 minutes
Format: Individual exercise

1. You are responsible for building a form for a corporate support website. It will capture customer details and the details of their issue.
2. Open the file `Ch03\Forms\form-exercise.html`.
    a. Complete the TODO items.
    b. Don't worry about the appearance. Just concentrate on meeting the requirements.
3. Test your form and check that the results page shows the values you expect.

Bonus Exercise (to be attempted if time permits)

4. Work on the performance of the form. Try to make it match the solution or at least line-up items in each group.

Chapter 4: Client-Side JavaScript Programming

Exercise 4.1: Jumping JavaScript

Time: 5 minutes
Format: Individual hands-on exercise

1. View the web page `Ch04\JavaScript\js-song.html` with your browser.
2. View the page source.
    a. What happens from the time that the page is loaded?
    b. Can you explain the resulting web page?

Exercise 4.2: JavaScript Arrays and Objects

Time: 15 minutes
Format: Individual hands-on exercise

1. View the web page `Ch04\JavaScript\objects.html` with your browser.
2. View the page source.
3. Answer the following questions:
    a. What happens from the time that the page is loaded?
    b. Can you explain the syntax of the script?
    c. Can you explain the logic? What is the advantage of organizing the varying data into arrays and objects?
4. Try putting a setTimeout around your alert with time set to zero as in the snippet below:

```
song += 'No more zipcodes jumping on the bed!<br>';
document.write(song);
setTimeout(function() {
    alert('Scan? Meter? Rhyme? What are those?');
    }, 0);
```

    a. What happens now?

Exercise 4.3: Manipulating the DOM

Time: 10 minutes
Format: Individual hands-on exercise

1. View the web page `Ch04\ChangeDOM\change-dom.html` with your browser.
2. View the HTML page and its associated JavaScript file in VSC.
3. Modify the web page to include a new `div` section.
4. In the JavaScript file, define a new function that displays a positive, uplifting message of the day in the new `div` element that you just defined on the web page.
5. Modify the web page so that your new function will be called when the page is loaded.
    a. Your message should then be displayed when the page is viewed.
6. How can you have both the original and your new functions called when the page is loaded?

Exercise 4.4: Working with Built-In Classes

Time: 15 minutes
Format: Individual hands-on exercise

1. View the web page `Ch04\Random\random.html` with your browser.
2. Notice that the image cycles as it did before, but this time the images are presented in a random sequence.
3. View the HTML page and its associated JavaScript file in VSC.
4. Create a new page that randomly selects an inspirational message and displays it on your page.
5. If you have time, display the date along with the message. Display the date in a foreign format or convert it to UTC.

Exercise 4.5: Responding to Events

Time: 15 minutes
Format: Individual hands-on exercise

1. View the web page `Ch04\EventHandler\one-by-one.html` with your browser.
2. Open the HTML and its associated JavaScript file in VSC.
3. Modify the `addEventHandler` function to register an event handler for the `mouseout` event.
4. Verify the web page performs as expected.
5. Try the `click` event instead.

Exercise 4.6: Form Validation with JavaScript (Optional)

Time: 20 minutes
Format: Individual hands-on exercise

1. View the web page `Ch04\Forms\form-exercise.html` with your browser.
    a. Try submitting the form with a "to" date that is later than the "from date".
    b. What happens?
2. Open the HTML in VSC.
3. Add validation to ensure the dates are the right way around.
4. There are two main approaches (shown below). Choose one and follow the instructions for that section.
    a. Put validation in the click event of the button.
    b. Put validation in the submit event (and optionally in the change event).
5. While working, you may wish to assign an event handler to the submit event that just calls `event.preventDefault()` or remove the action from the form. Make sure to remove this code when you have it working.

Validation in the Click Event

6. This will use the built-in form validation.
7. Create a method that gets the two values from the form and compares them.
    a. Convert the values to `Date` using the simple constructor that accepts a string (normally, use of this constructor is deprecated due to browser incompatibilities, but in this case, we will just compare two values constructed in the same way and it will be safe).
    b. Compare the dates. If the dates are in the wrong order, `setCustomValidity` on *one* of the two date input controls.
    c. Assign this method as the event handler of the button's `click` event.
8. In the `input` event of *both* date input controls, call `setCustomValidity` with an empty string to reset the valid state.

Validation in the Submit Event

9. This will use a custom message field to display the validation message.
    a. Create the custom message under the rest of the form controls.
    b. It should start out in a hidden state.
    c. The stylesheet contains classes `error`, `spancol`, and `hidden`, which you may find helpful, but are not obliged to use.
10. Create a method that gets the two values from the form and compares them.
    a. Convert the values to `Date`. See the comment in the previous section.
    b. Compare the dates and, if they are the wrong way around, reveal the error message and return `false`.
    c. Otherwise, return `true`.
11. In the `submit` event handler for the form, call the validation method and call `preventDefault()` if validation fails.
12. You also need a method of hiding the error message when it no longer applies.
    a. You could do that in the validation function, but the error message would remain showing until you pressed the submit button.
    b. The obvious place is in the `change` event of *both* date input controls.
    c. However, it is possible to change the value to another invalid value: the message would disappear until the submit button is pressed. Better to hide the message and call validation, so the message remains displayed if the values are still invalid.

This page intentionally left blank.

Chapter 5: Working with jQuery

Exercise 5.1: First Steps with jQuery

Time: 15 minutes
Format: Individual hands-on exercise

1. Using VSC, create a new web page named `hide-and-seek.html`.
2. Add a script element for jQuery. You can copy a link from the jQuery CDN: https://code.jquery.com/.
    a. Use the current minified link.
    b. Place it directly under title tags in your head section of your html page.
3. Add multiple `<button>` and `<div>` elements to your page.
    a. Do NOT place the `<button>` and `<div>` elements inside a `<form>` tag. The button would default to a submit button, which is NOT what we want.
4. Create a new JavaScript file named `hide-and-seek.js` and load it in the web page.
5. In the JavaScript file, define event handlers using jQuery.
    a. Have one of the buttons hide all the `<div>` elements.
    b. Have another button that shows all the `<div>` elements.
    c. Bind the event handlers to the events in the jQuery `ready` function.
6. Verify that your buttons work as expected.

Bonus Exercise (to be attempted if time permits)

7. Experiment with other functions from the jQuery effects category (https://api.jquery.com/category/effects/).
    a. Consider, for example, `fadeToggle()`, `toggle()`, and `slideToggle()`.

Exercise 5.2: Putting It All Together

Time: 30 minutes
Format: Individual or pair-programming hands-on exercise

1. View the web page `Ch05\Forms\form-exercise.html` with your browser.
    a. Try submitting the form with a "to" date that is later than the "from date".
    b. The form currently permits it.
2. Open the HTML in VSC.
3. Implement validation to ensure the dates are the right way around. Do this as far as possible using jQuery.
4. While working, you may wish to assign an event handler to the submit event that just calls `event.preventDefault()` or remove the action from the form. Make sure to remove this code when you have it working.
5. Create a validation method that gets the two date values from the form and compares them.
    a. Convert the values to `Date` using the simple constructor that accepts a string (normally, use of this constructor is deprecated due to browser incompatibilities, but in this case, we will just compare two values constructed in the same way and it will be safe).
    b. Compare the dates. If the dates are in the wrong order, `setCustomValidity` on *one* of the two date input controls.
    c. Assign this method as the event handler of the button's `click` event.
6. In the `input` event of *both* date input controls, call `setCustomValidity` with an empty string to reset the valid state.
    a. Note that `setCustomValidity` is not available as a jQuery method, so you will need to access the raw HTML Element.

Bonus Exercise (to be attempted if time permits)

7. Add code so that the form's title changes color when the mouse pointer is over the submit button.

Exercise 5.3: Ajax with jQuery

Time: 30 minutes
Format: Individual or pair-programming hands-on exercise

Start the SimpleServer

1. Open a command window in the `Ch05\SimpleServer` folder.
2. Install all the SimpleServer dependencies by typing the following command in the command window:

   Be sure to change the directory of your terminal window to `Ch05/SimpleServer` before running `npm install`.

   ```
   npm install
   ```

3. It is OK to fix any issues that are mentioned by running:

   ```
   npm audit fix
   ```

4. Start the SimpleServer by typing the following command:

   ```
   npm start
   ```

5. Once the server has started and the message indicating that the server is listening, you may proceed to complete the Ajax client application.
6. In a browser window, open the index page of the service (`localhost:3000`) and check the URLs links or routes.

Complete the Ajax Client

7. Open the file `Ch05\Ajax\ajax_exercise.html` for reviewing in VSC:
   a. When completed, this page will have a list of contact groups. When the user selects a contact from the group, it will retrieve the list of contacts in that group.
   b. It contains an empty `SELECT` element that we must populate with `OPTION` elements.
   c. If you are not comfortable with `SELECT`s, review the documentation (e.g., https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select).
8. Open the file `Ch05\Ajax\ajax_exercise.js` for editing.

9. Start by retrieving the list of groups.
    a. You should be able to find a URL supported by the service that returns a list of groups. Make sure you are familiar with what that call returns.
    b. Complete the steps marked `TODO 1`.

       Hint: `http://localhost:3000/groups`

    c. You may wish to log the data returned from the service before attempting to add it to the `SELECT`.

       Hint: `console.log('my stuff here ' + selection);`
10. Verify that when the web page opens, it sends an Ajax request and creates a `SELECT` using the group list that the SimpleServer returns.
11. Now add the functionality to retrieve contacts from the server.
    a. Find a URL supported by the service that retrieves contacts from a group.
    b. Complete the steps marked `TODO 2`.
12. Verify that the web page functions as expected.

Bonus Exercise (to be attempted if time permits)

13. Change the code to display the results in a table.

Chapter 6: Introduction to Angular

Exercise 6.1: Getting Started with Angular

Time: 20 minutes

In this exercise, you will create a new Angular application and view it in the Google Chrome browser. You will make some simple changes to the application and verify that the changes are immediately displayed in the browser.

1. Using Windows File Explorer, copy `M:\FSE\WebApps\WebAppsClient.zip` to `D:\`.
2. Right-click **`D:\WebAppsClient.zip`** and select **7-Zip > Extract Here**.
3. From the VS Code menu bar, select **File > Open Folder**. Select **`D:\WebAppsClient\BookStore`**.
    a. If VS Code asks if you trust the authors, click the checkbox **Trust the authors of all...**, then click the **Yes, I trust...** button.
4. Explore the `BookStore` folder in VS Code and examine the file and folder structure.
    a. This is a template Angular application that was generated with the command `ng new`.
5. Click the file `package.json` to open it in the editor. Among other things, this file contains a list of `npm` packages required by the application. It also defines some `npm` scripts (e.g., `npm start` executes `ng serve`).
6. From the VS Code menu bar, select **Terminal > New Terminal**
    a. Verify the terminal's current directory is `D:\WebAppsClient\BookStore`.
7. Run the application by entering the following command. *Note:* this command is very slow the first time, so run the command and then move on to the next step:

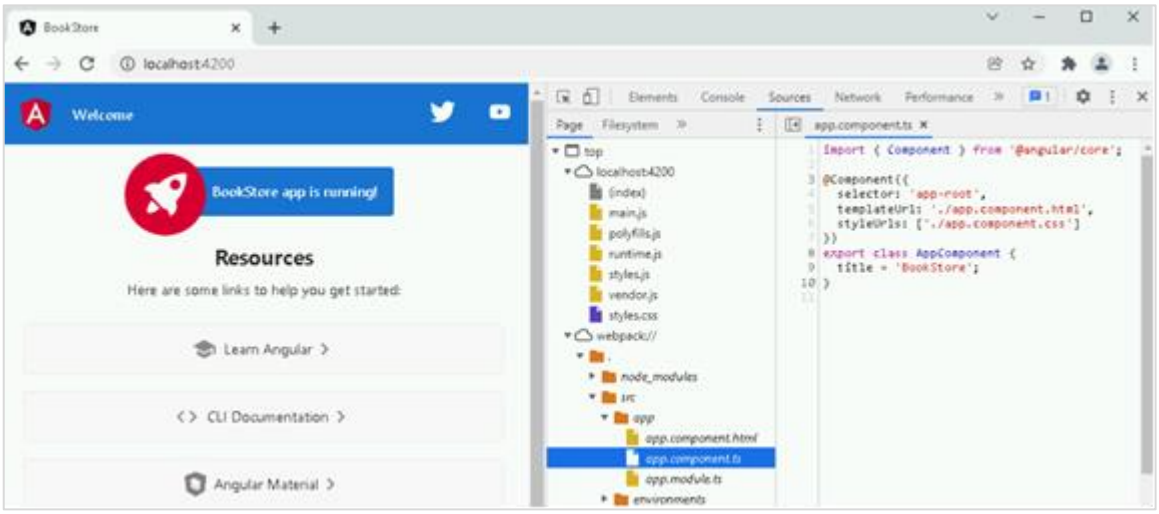
```
npm install

npm start
```

8. Set up `git` by running the following commands. The VS Code extension **Git History** is already installed, so you can revert mistakes, and it's easy to review your changes in each exercise:

```
git init
git add -A .
git commit -m "Initial commit"
```

9. Open Google Chrome and navigate to `http://localhost:4200`. You might (briefly) see the text **Loading...** before `BookStore app is running!` appears.

10. Right-click **Welcome** in the browser and select **Inspect**. The development tools will open. Click the **Sources** tab, and then expand the **webpack | src/app** nodes. Then click the file **app.component.ts**, or the file starting with that name. You should see something like this:



You can debug your code here. Clicking next to the line number sets a breakpoint.

11. Return to VS Code, navigate to the file `app.component.ts` in the `src/app` folder, and open it in the code editor. Change the title to be `Angles on Books`.

12. Save the file. It might already have been saved if you selected Auto Save in VS Code's File menu.

13. You should now see messages in the terminal/command window as the `webpack` build tool invalidates and recompiles the bundle. Wait for the process to complete, and then return to your browser. You should see that the text on the page, and the source in the development tools, have both changed.

    Note: You may need to click **Play** to debug.

14. Now let's see what happens when there's an error. Return to VS Code, delete the `'` (single quote) at the end of the `templateUrl` string, and save your work. You should see immediate syntax error reporting in the terminal window. Return to the browser window and examine the syntax error reporting in the **Console** tab of Google Chrome's development tools.

15. Fix the error by replacing the `'` character, then note the results in both the terminal window and Google Chrome.

    Usually, you can leave Angular's live compiler running. Sometimes it gets confused by a serious problem or a structural change. If you no longer see it re-compiling automatically, stop it by pressing CTRL+C and say "`y`" to stop the batch file, then restart it.

16. Shut down the server by typing CTRL+C in the terminal window.

Bonus Exercise (to be attempted if time permits)

Work on the **Getting started with Angular** tutorial at https://angular.io/start.

Exercise 6.2: Write Your First Test Specs

Time: 20 minutes

In this exercise, you will write tests for a piece of pre-written JavaScript code.

If the server from a previous exercise is still running, shut it down by typing CTRL+C in the terminal window.

1. Open a terminal window for the `Ch06\JasmineExercise` directory.
2. Run the following commands in the terminal window:

```
npm install
```

```
npm install jasmine --save-dev
```

3. Open the file `Ch06\JasmineExercise\MathUtils.js` with VSC.
4. Examine the code in this file.
    a. This is the code we want to test.
5. Open the file `Ch06\JasmineExercise\spec\MathUtils.spec.js` with VSC.
6. Examine the code in this file.
    a. This is the test file that you need to complete.
7. Complete the TODO steps in this file.

    Hint: For an exception that throws a new Error:

```
it("throws an error when factorial's argument is negative", () => {
    expect(() => {
        calc.factorial(-7);
    }).toThrowError(Error);
});
```

8. Verify that all the tests pass. To run your tests, run the following command in the terminal window:

```
npm test
```

9. Add more tests for the basic Math functions in the `MathUtils.js` file.
    a. Consider both success (positive) and failure (negative) conditions.

Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

Exercise 6.3: Unit Testing Angular

Time: 20 minutes

In this exercise, you will use Karma and Jasmine to create automated unit tests for an Angular application.

1. In Visual Studio Code, reopen the folder for the Angular `BookStore` app you worked on earlier. (Or you can modify the solution to Exercise 6.1; ask your instructor for the solution project.)
2. Open `app.component.spec.ts` from the `src/app` folder and examine the contents.
    a. This elementary test class worked with the original version of `app.component.ts` and `app.component.html`.
3. Run `npm test` in a terminal window. This will launch the Karma test runner and run all of the tests defined in `app.component.spec.ts`. There should be failing tests at this point.
4. Examine the resulting error message that will be displayed in the browser.
    a. Your test will fail at this point because you have not changed the title that the test is expecting.
    b. The error messages are very specific as to the source of the errors.
5. Make the necessary changes to get all the tests to pass.
    a. If you run into problems, work with your teammates or your instructor to get your unit tests to pass.

Bonus Exercise (to be attempted if time permits)

6. The Compodoc documentation tool will automatically generate documentation for your Angular project. Try it out:
    a. View the Compodoc's project's website at https://compodoc.app/.
    b. Install Compodoc: `ng add @compodoc/compodoc`
    c. Generate documentation: `npm run compodoc:build-and-serve`
    d. View the documentation by browsing to `http://localhost:8080/`.
    e. You can customize the documentation by adding JSDoc comments to your classes, methods, and properties. See https://compodoc.app/guides/jsdoc-tags.html.
        i. Add a JSDoc comment directly above your `AppComponent` class definition:

```
/**
 * AppComponent is the root component of the application.
 */
```

        ii. Restart the `compodoc` command and refresh the `AppComponent` documentation page to see the output of the JSDoc comment

Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

This page intentionally left blank.

Chapter 7: Angular Components

Exercise 7.1: Creating a Component

Time: 30 minutes

In this exercise, you will create a new component representing a `BookList` and its properties. You will also create a new model class representing a `Book` and its properties.

1. Open the `BookStore` application directory in Visual Studio Code. Verify that all tests run and pass by running `npm test` in a terminal window. Fix if needed. All tests must pass before starting this exercise.
2. In a second terminal window, start the application by running `npm start`. Verify the application is running as expected by visiting `localhost:4200` in your favorite browser.
3. You may decide how you wish to work with the live compiler: you may find that some changes break the compilation process, and you will need to stop the compiler with CTRL+C and then restart it, but most changes should work well.
    a. If you wish to leave the live compiler running, open a new command window in the `BookStore` directory and use this to issue the ng commands below. If you take this option, you will have three terminal windows: the live compiler, karma testing, and the interactive window for ng commands.
    b. You can periodically stop the compiler from issuing `ng` commands and then restart it by typing `npm start`.
4. We want to create a class to carry data about our books. We will store all these model (or domain) classes in a folder together to keep them separate from our Angular components:
    a. Use the Angular CLI:

    Reminder: Be sure you are in the correct directory before using the CLI.

    ```
    ng g class models/book
    ```

5. Use this file to define a class `Book` with a `constructor` accepting four `public` arguments: `title`, `author` and `cover` of type `string`, and `bookId` of type `number`. Make sure the class is exported from the file. The code is below. (***Note:*** this will cause the Book spec to fail. You'll fix that in the next step.)

```
export class Book {
    constructor(
        public title: string,
        public author: string,
        public cover: string,
        public bookId: number) { }
}
```

6. Correct the error in the `Book` test spec.
    a. Use a Book constructor with four arguments in the test.

    Your code will look something like this:

```
describe('Book', () => {
      it('should create an instance', () => {
      expect(new Book("Think and Grow Rich","Napoleon Hill",  "",42))
          .toBeTruthy();
          });
});
```

7. Next, create `BookPageComponent`:
    a. Type the following command in a command prompt:

    ```
    ng g component books/BookPage
    ```

    In VS Code, note that a new folder named `books` has been created, and another folder, `book-page`, has been created inside it. We will put all our book-related Angular components in a common folder called `books`. Later, we will see the significance of this folder, but for now, it will just be a way to organize our code

    b. Look at the `book-page.component.html` and note that it says `<p>book-page works!</p>` for now. Eventually, you will see that on the page.

8. Review the Karma output to confirm that your code still passes all the tests.
    a. You may need to include the book component to make all tests pass.

9. Inside the `BookPageComponent` class, add a `book` property of type `Book` and assign it to an object literal defining all four properties of the `Book` class.
    a. Set the `cover` to an empty string literal, the `bookId` to `1`, and the `title` and `author` properties to any book you like.

b. You will need to `import` the `Book` type from the `models` folder. Investigate the assistance that VS Code can give you: click the red-underlined `Book` and click the lightbulb that appears. Select the option that imports the file with a relative path from the menu that appears. If there is no option with a relative path, modify the path to be relative.

c. Your component should look a little like this:

```
import { Component, OnInit } from '@angular/core';
import { Book } from '../../models/book';

@Component({
    selector: 'app-book-page',
    templateUrl: './book-page.component.html',
    styleUrls: ['./book-page.component.css']
})
export class BookPageComponent implements OnInit {

    book: Book = {
        title: 'The Lord of the Rings',
        author: 'J R R Tolkien',
        cover: '',
        bookId: 1
    };
    constructor() { }

    ngOnInit() {
    }
}
```

10. Open the `app.module.ts` file.

    a. Note that `BookPageComponent` has been added to the declarations array and that an import statement has been added.

11. Open `app.component.html`.

    a. Delete everything in the file.

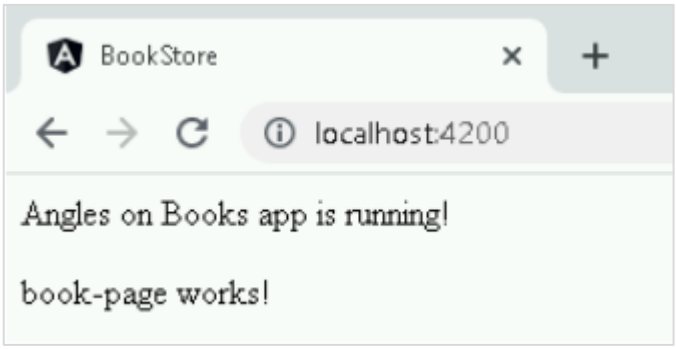
    b. Add a heading that displays the `AppComponent`'s title property:

    ```
    <h1>Welcome to the {{title}}!</h1>
    ```

    c. Add the custom element for the `AppBookPageComponent`:

    ```
    <app-book-page></app-book-page>
    ```

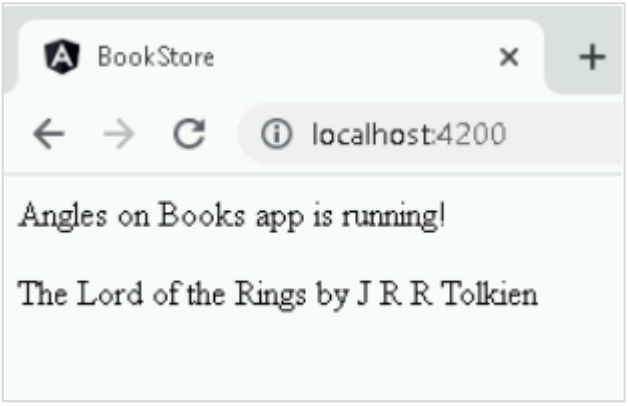
12. Run the application and check that the output is as you expect. It might look like this:



13. If you weren't already, run the application tests. Do they pass? Fix them, so they do.
    a. You have added a component to `AppModule`. Look at `app.component.spec.ts`: it also creates a module (the testing module). You may need to add the same component to the declarations array of the testing module and check that VS Code has automatically added a file import.
    b. Note that Karma will show a failing test because of an Angular error: "app-book-page is not a known element". You'll fix that in the next step.
    c. We will suggest some tests throughout these exercises, but our suggestions are very far from full TDD. Treat them as inspiration!
14. Now add a placeholder for the book in the component template.
    a. We will tidy this up in the subsequent exercises, but for now, just add interpolation bindings for the book's title and book's author.
    b. Replace the "book-page works" text in `book-page.component.html`. It might look something like this:

```
<p>
    {{book.title}} by {{book.author}}
</p>
```

c. Check that your page displays as you expect.



Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

Exercise 7.2: Unit Testing a Component

Time: 20 minutes

In this exercise, you will write unit tests for the `AppComponent` and the `BookPageComponent`. A good unit test for an Angular Component will not depend on any other Angular Components.

For the `AppComponent` unit test, you will create a `MockBookPageComponent` to ensure sound test isolation between the root component and the new book page component.

Then, proceeding in true TDD style, you will write a unit test to verify that the `BookPageComponent` is working correctly.

Then you will implement the layout in the HTML template for the `BookPageComponent` to pass the test.

1. **AppComponent unit test**: In the last exercise, we updated the test module in the `AppComponent` test script to reflect the new dependency we had added (we had added a dependency on `BookPageComponent`). We will add a mock to isolate these two components properly.
    a. Open `app.component.spec.ts`.
        i. You may need to remove the file import for the `BookPageComponent`.
    b. Declare a local mock of `BookPageComponent` that does nothing.
    c. You may need to import `Component` from `@angular/core`.
2. Run the tests to verify that the AppComponent unit test passes. We will use this mock later.

3. **BookPageComponent unit test**: Next, we will change the `BookPageComponent` to write the book into a table. First, write a test that checks whether the output has been rendered correctly. Writing tests that inspect the DOM is complex (see the example on the next page), and it will take some experience before you can write them accurately ahead of making your changes.
    a. If you like, you can initially write your test without expects. Just logging the DOM will give you some experience of how the DOM changes when you change a template. Even if you don't write a test using TDD, it will still protect you from regression bugs.
    b. Open `book-page.component.spec.ts`.

c. Add an extra test that uses the fixture to inspect the DOM. Run the test and watch it fail.

d. Here is one way. We have included a log statement so you can see what the table looks like.

```
it('should contain a table', () => {
  const compiled = fixture.debugElement.nativeElement;
  const table = compiled.querySelector('table');
  console.log(table);
  expect(table.rows.length).toBe(1);
  expect(table.rows[0].cells[0].textContent)
   .toBe('The Lord of the Rings');
});
```

4. Now, add the table to the template inside `book-page.component.html`. Replace the placeholder text with a `table` with a single row, and use data bindings for `book.title` and `book.author`. Save your work, check the results inside Google Chrome, and check that the tests pass (remember to check that the tests are compiling since sometimes structural changes cause the test compilation to crash).

```
<table>
    <tr>
        <td>{{book.title}}</td>
        <td>{{book.author}}</td>
    </tr>
</table>
```

Run your tests again and verify they pass.

Hint: To make the table more easily visible, try setting the border to a width of 2px solid black by modifying the `book-page.component.css` file.

```
table, th, td {border: 2px solid black;}
```

Bonus Exercise (to be attempted if time permits)

5. Improve the HTML in the template by adding `<thead>`, `<tbody>` elements. Inside the `<thead>`, add a new table row and `<th>` elements with the text `Title` and `Author`. Before you make the changes, take a look at the test and try to make appropriate changes so that it passes after you have changed the table.
6. Improve the look and feel by adding an `<h2>` element above the table with the content `Books`.
7. You could also consider defining and using CSS styles to control the styling of the book list.

Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

Exercise 7.3: Using Built-In Directives

Time: 20 minutes

In this exercise, you will convert the book property into an array of Books and display it in a table using `*ngFor`.

1. First, we need to update the test that looks at the table's contents in the Book Page Component since it needs to deal with multiple data rows.
    a. We will be hardcoding our list of books, for now, so you can choose a number of rows. We suggest two data rows (plus header row if you did the optional section of the last exercise). That's enough to prove that the functionality works but doesn't require you to create a lot of test data.
    b. Open `book-page.component.spec.ts`.
    c. Change your test to expect the appropriate number of rows in the table.
    d. The test will fail because the table doesn't have this number of rows yet.
2. Create the `books` property in the `BookPageComponent`. The property is an array of `Book`. Replace the existing `book` property with an array and extend the current initialization.

```
books: Book[] = [{
    title: 'The Lord of the Rings',
    author: 'J R R Tolkien',
    cover: '',
    bookId: 1
}, {
    title: 'The Left Hand of Darkness',
    author: 'Ursula K Le Guin',
    cover: '',
    bookId: 2
}];
```

3. Open the `book-page.component.html` file and add an `*ngFor` directive to the table row that displays book data. Individual books should be assigned to a local `book` variable.

```
<tr *ngFor="let book of books">
```

4. Now the test should pass. Keep working on the test and the code until it does. You can see your test output in the Karma window.

Bonus Exercise (to be attempted if time permits)

5. Try other directives such as `*ngIf`, `ngClass`, and `ngStyle`.
    a. For example, you could replace the table with some suitable text if there are no items in the list of books.
    b. To test this, you will need to set the book list to be empty:

```
component.books = [];
fixture.detectChanges();
```

Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

Exercise 7.4: Refactoring Components

Time: 45 minutes

In this exercise, you will refactor the `BookPageComponent` into two components: a simple component to display any list of books that we will call `BookListComponent`, and the remainder of the existing `BookPageComponent`, which will manage the book list and contain any book-related components. `BookListComponent` will be reusable anywhere we need a list of books, and it will receive a list of books through an `@Input` binding.

1. Make sure that you are in the `BookStore` folder in the integrated terminal. Using the CLI, create a new component named `book-list` in the `books` folder by running the following command:

   ```
   ng g component books/BookList
   ```

2. At times adding components causes problems with tests. Stop the tests by pressing CTRL+C in the integrated terminal running the tests to be safe.
3. Move any tests for the table contents to the new `BookList` component.
   a. If you created a `DIV` to be displayed when there are no items in the list, move tests for that as well.
   b. The new component will not have a hard-coded list of books. It will always receive them from a parent component. You will need to create a list of books in the test(s) that check for them and assign that to the `books` property of the component.
4. Start the tests again in its integrated terminal.
5. In `book-list.component.ts`, create a `books` property and decorate it with an `@Input` decoration.
   a. Initialize it to an empty array and add the required imports.
   b. If you created the `trackBy` method, move that from `book-page.component.ts`.
6. Move all the HTML that involves the table of books from `book-page.component.html` to `book-list.component.html`.
   a. As with the tests, if you created a `DIV` displayed when there are no items in the list, move the mark-up for that.

7. Move any styles that apply solely to the book list from `book-page.component.css` to `book-list.component.css`.
8. At this point, all your tests should pass, but there is no longer any table displayed.
9. Add the selector for `BookListComponent` to the template for `BookPageComponent` and set up the property binding.
    a. Start by adding a mock `BookListComponent` to `book-page.component.spec.ts`. This mock needs an `@Input` property to match the real `BookListComponent`: in other words, it must match the public interface of the component.

```
@Component({
    selector: 'app-book-list',
    template: 'mock book list'
})
class MockBookListComponent {
    @Input()
    books: Book[] = [];
}
```

    b. Add the mock `MockBookListComponent` to the declarations section of the `TestBed.configureTestingModule`.

```
beforeEach(async () => {
        await TestBed.configureTestingModule({
            declarations: [
                BookPageComponent,
                MockBookListComponent
            ]
        })
            .compileComponents();
```

    c. In `book-page.component.spec.ts`, create a test that checks the books property is passed to the child component.

```
it('should pass books to the child component', () => {
    const bookList = fixture.debugElement.query(
        By.css('app-book-list')).componentInstance;
    expect(bookList.books.length).toBe(2);
});
```

d. Modify the template for the `BookPageComponent`:

```
<h2>Books</h2>

<app-book-list [books]="books"></app-book-list>
```

10. We have now separated the book list functionality from the book page and isolated the tests. The book page contains very little apart from the list of books, but that will change over the following exercises.

Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

This page intentionally left blank.

Chapter 8: Angular Modules and Binding

Exercise 8.1: Creating a Module

Time: 20 minutes

In this exercise, you will create a separate `Books` module and add it as an import in the main module `AppModule`.

Reminder: The Angular live server usually restarts automatically as you save your changes, but occasionally you may need to stop and restart the `npm test` and `npm start` commands.

1. Create a Books module by executing the following command:

   ```
   ng generate module Books
   ```

   a. Notice that the module is put in the `books` folder. We would typically have added all the related components to the module from the beginning, and that was why we grouped them in a folder.

2. Add the following to the declarations array in `books.module.ts`:

   a. `BookListComponent`

   b. `BookPageComponent`

   c. VS Code should fix the file imports automatically. If it doesn't, add them manually.

3. Add an `exports` property to the object literal that is passed as the argument to `@NgModule`. The value of the `exports` property should be an array with one item, `BookPageComponent`.

4. Modify `app.module.ts` to use the `BooksModule` rather than declaring the various components directly.

   a. Remove `BookListComponent` and `BookPageComponent` from the declarations array and the file imports.

   b. Add the `BooksModule` to the imports array. Let VS Code fix the file import for you.

5. Check whether the application runs and the tests pass.

   a. None of the tests depend on the module (it is just a collection of components), so they should all pass.

Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

Exercise 8.2: Two-Way and Event Binding

Time: 60 minutes

In this exercise, you will use two-way bindings and communicate with a parent component using an Output binding. You are going to add a new component containing HTML input controls. The inputs will use two-way binding to add new books to your existing books array.

Important Note: Confirm that you are in the correct directory before running CLI commands. The current directory must be `BookStore` (not `BookStore/src`, `BookStore/src/app`, etc.)

1. Using the CLI, create a new component named `book-form` in the `books` folder. You could explicitly add it to the `BooksModule` using the option `-m=Books`, but that isn't necessary because Angular will infer that from the location:

   ```
   ng g component books/BookForm
   ```

2. Open the `books.module.ts` file.
   a. Verify that the `BookFormComponent` is imported.
   b. Verify the `BookFormComponent` has been added to the declarations array attributes in the declarator.
   c. Add a file import for the `FormsModule` from `@angular/forms`.
   d. Add `FormsModule` to the imports array.

3. Open `book-page.component.html` and add the book form selector above the book list selector.

   ```
   <h2>Books</h2>

   <app-book-form></app-book-form>
   <app-book-list [currentBooks]="books"></app-book-list>
   ```

4. Check that the book form placeholder is correctly displayed by viewing `localhost:4200` in your favorite browser.
    a. The text "book-form works" should now appear above the table. You will be adding more relevant content to this soon. Right now, this is just a placeholder where content will be displayed in the future.
5. Do your tests pass?
    a. Once again, you added a dependency to the book page, so `book-page.component.spec.ts` will need some changes.
    b. Add a mock of the `BookFormComponent`, named `MockBookFormComponent`:

```
@Component({
    selector: 'app-book-form',
    template: ''
})
class MockBookFormComponent {
    book: Book = new Book('', '', '', -1);
    @Output() createBook = new EventEmitter<Book>();
    add() {
        this.createBook.emit(this.book);
    }
}
```

    c. Add the mock `MockBookFormComponent` to the declarations section of the `TestBed.configureTestingModule`.

```
beforeEach(waitForAsync(() => {
        TestBed.configureTestingModule({
            declarations: [
                BookPageComponent,
                MockBookListComponent,
                MockBookFormComponent
            ]
        }).compileComponents();
    }));
```

6. We are now going to complete the Book Form Component. It is usually easiest to start by creating the template and use that to determine what capabilities the component needs.

7. Open the `book-form.component.html` file and modify it so that it contains a `div` above the table of books that has two `text` inputs. Use labels to name the two inputs `Title` and `Author`, and add a button `Add Book` of type `button`. Disable the button unless both the title and author inputs are populated. *Note:* there will compile errors on the references to the non-existent `book` property.

```
<div>
    <label for="title">Title:</label>
    <input type="text" id="title" />
    <label for="author">Author:</label>
    <input type="text" id="author" />
    <button type="button" [disabled]="!book.title || !book.author">
        Add Book</button>
</div>
```

8. Add the `book` property to `BookFormComponent`:
    a. Create a public property `book` of type `Book` in `BookFormComponent`, and assign it to a `new Book()` object. Use three empty string literals and `-1` as arguments to the constructor.
    b. Add two-way bindings for `ngModel` to each of the `input` elements in the `book-form.component.html` file, binding the values to the appropriate properties of the `book` object as illustrated in the course notes.

    Hint:

    ```
    <label for="title">Title:</label>
    <input type="text" id="title" [(ngModel)]="book.title" />
    ```

9. Your code should still compile and run, even though it doesn't do anything yet. But your tests will fail. Remember when we added `FormsModule` to the `imports` array of the `BooksModule`?
    a. Did we add it to the tests?
    b. Fix that. You will need to edit `book-form.component.spec.ts` and add the `imports` array to the object literal argument to `TestBed.configureTestingModule()`, then add `FormModules` as the one item in the `imports` array.
10. At this point, your application should run, and your tests should pass. The application doesn't do anything when you type in the input fields, and there are no tests for the inputs.
    a. What type of testing could you apply to the input fields?
    b. Should you? There's limited value in testing whether Angular works, so we won't add any tests for the input fields.

11. Add the button method:
    a. Create a new `add()` method in `BookFormComponent`. For now, have it log the current value of the book to the console. You will need to refer to the property as `this.book`. (Note that `ng serve` doesn't log to the terminal console, but messages are visible in Chrome > F12 > Console).
    b. Use event binding in the `book-form.component.html` file to bind the `add()` method to the button's `click` event.
    c. We would like our method to add the new book to the array, but that is held in the Book List Component, and the book form has no access.

12. At this point, your `BookFormComponent` should look a little like this:

```
import { Component, OnInit } from '@angular/core';
import { Book } from '../../models/book';

@Component({
    selector: 'app-book-form',
    templateUrl: './book-form.component.html',
    styleUrls: ['./book-form.component.css']
})
export class BookFormComponent implements OnInit {

    book: Book = new Book('', '', '', -1);

    constructor() { }

    ngOnInit() {
    }

    add() {
        console.log(`book = ${JSON.stringify(this.book)}`);
    }
}
```

13. Check that your application works. Enter some values and see them logged to the console when you press the button. Check that your tests still pass.
14. To pass the book object to a handler in the `BookPageComponent`, you will create an `@Output` property and emit an event from the `BookFormComponent`.
    a. To accomplish this, you will need to import `EventEmitter` and `Output` from `@angular/core`.

       Hint: Do not import `EventEmitter` from stream. Import from `@angular/core`.

b. Start by declaring the event emitter in `BookFormComponent`. Name it `createBook` and decorate it with `@Output()`. You should parameterize the `EventEmitter` with `Book` since it will be passing a book.

```
@Output()
createBook = new EventEmitter<Book>();
```

c. Create a test in `book-form.component.spec.ts` to spy on the `emit` event of your EventEmitter.

d. Your code will look something like this:

```
it('should emit an event on click', () => {
    spyOn(component.createBook, 'emit');

    // trigger the click
    const nativeElement = fixture.debugElement.nativeElement;
    const button = nativeElement.querySelector('button');
    button.dispatchEvent(new Event('click'));

    fixture.detectChanges();

    // check the output event was triggered
    expect(component.createBook.emit).toHaveBeenCalled();
});
```

This test will be red (as it should be) until you add the production code in the step below. (Think Red, Green, Refactor.)

e. In the `BookFormComponent`, revise the code inside the `add()` method to `emit` the `createBook()` event, passing in `this.book` as the sole argument.

f. The tests should all pass at this point.

15. Now, create the handler in `BookPageComponent`.

a. Create a test for an `addBook` method. The test should call `addBook`, passing in a book object and then check that the book is added to the `books` array.

```
it('should add a book to the array', () => {
    const oldLength = component.books.length;
    component.addBook(new Book('The Lathe of Heaven',
        'Ursula K Le Guin', '', 3));
    expect(component.books.length).toBe(oldLength + 1);
    expect(component.books[oldLength].title)
        .toBe('The Lathe of Heaven');
});
```

b. The `addBook` method doesn't exist, so create a placeholder for it in `book-page.component.ts` that accepts a single argument of type `Book`.

c. The test should fail since the `addBook` method does not do anything yet.

d. Implement the `addBook()` method in `book-page.component.ts`. The method should `push` the book into `this.books`. It should not return anything.

e. Finally, use a method binding inside the `book-page.component.html` template to assign `addBook($event)` to the `app-book-form` element's `createBook` attribute.

```
<app-book-form (createBook)="addBook($event)"></app-book-form>
```

16. Check that your application works and all the tests pass.

    Note: The above test adds but does not remove a book. In a real app, this test might add books to the database, which could be an issue. The basic test logic is good.

17. Check that your application works and all the tests pass.

18. However, there is a problem.

    a. Add a book to the list.

    b. Now change the contents of the input fields.

    c. What happens?

19. We need the `add()` method to create a new Book after emitting the event.

```
add() {
    this.createBook.emit(this.book);
    this.book = new Book('', '', '', -1);
}
```

20. Improve the presentation of the form as best you can.

    a. Add a `<footer>Copyright Fidelity</footer>` in an appropriate location.

    b. Consider using some other appropriate HTML semantic tags.

    c. Define a CSS style in `book-page.component.css` to make the `<h1>` elements blue.

    d. Be creative and define some other CSS styles and use them in `book-page.component.html`.

Bonus Exercise (to be attempted if time permits)

Continue working on the **Getting started with Angular** tutorial at https://angular.io/start.

This page intentionally left blank.

Chapter 9: Angular Services

Exercise 9.1: Creating and Injecting a Service

Time: 45 minutes

In this exercise, you will move the logic for retrieving and saving books into a service. You will create a new Angular service called `BookService` and move all logic relating to adding and retrieving books into that service.

1. Create the new service.

   ```
   ng generate service book
   ```

   Note: Creating this service may break your tests. If so, stop the tests and restart them.

2. Open `book.service.spec.ts`.
   a. You will need to import `inject`, `tick`, and `fakeAsync` from `@angular/core/testing` for the test you are about to write.
   b. `Book` needs to be imported from `./models/book`.
   c. You will write the `getBooks` method shortly.
3. Add a test for `BookService.getBooks`(). This method will return `Observable<Book[]>`, so the test needs to process that return asynchronously.
   a. The test won't compile because the `getBooks` method has not been written yet.

```
it('should return books', inject([BookService],
        fakeAsync((service: BookService) => {
    let books: Book[] = [];
    service.getBooks()
        .subscribe(data => books = data);
    tick();
    expect(books).toBeTruthy();
    expect(books[0].title).toBe('The Lord of the
Rings');
})));
```

4. Create the `getBooks` method.
    a. it should return `Observable<Book[]>`. And, for now, throw an error.
5. Now make the test pass by implementing `getBooks()`.
    a. Copy the `books` property from `BookPageComponent` and paste it into the `BookService` class.
    b. Now change getBooks() and have it return of(this.books).
6. Now move on to the `addBook()` method.
    a. This will mostly behave like `addBook()` in `BookPageComponent`, but it should return `Observable<Book>`. Remember that Observables must be subscribed to, or they do nothing.
    b. Start by creating a test. This is a little more complex.

```
it('should add a book', inject([BookService],
        fakeAsync((service: BookService) => {
    let books: Book[] = [];
    let added!: Book;  // The "!" allows "added" to be null
    const expected = new Book('A Wizard of EarthSea',
        'Ursula K Le Guin', '', 3);
    service.getBooks()
        .subscribe(data => books = data);
    tick();
    const expectedLength = books.length + 1;
    service.addBook(expected)
        .subscribe(data => added = data);
    service.getBooks()
        .subscribe(data => books = data);
    tick();
    expect(books.length).toBe(expectedLength);
    expect(books[books.length - 1]).toBe(expected);
    expect(added).toBe(expected);
})));
```

    c. Then, make the test pass by implementing the method. It should add the `Book` to the array and return an `Observable` from it. This last point may seem a little strange, but methods that add a new piece of data often also return it (usually with the id replaced by an assigned value).

7. You now have a working service. The advantage of testing the service separately is that any problems from here must be related to the way you are using it rather than the service itself.
    a. Your code for `BookService` may look a little like this:

```
@Injectable({
    providedIn: 'root'
})
export class BookService {
    books: Book[] = […];

    addBook(book: Book): Observable<Book> {
        this.books.push(book);
        return of(book);
    }

    getBooks(): Observable<Book[]> {
        return of(this.books);
    }

    constructor() { }
}
```

8. Now we will change `BookPageComponent` so it gets the list of books from `BookService` rather than having it hard-coded.
    a. For now, the `BookService` will have the hard-coded data.

9. We do not want the tests in `book-page.component.spec.ts` to depend on `BookService`, since they are unit tests. So, we will start by amending the tests so they provide a dummy service using Jasmine's Spy capability.

   a. Open `book-page.component.spec.ts`. At the start of the `beforeEach()`, before configuring the testing module, create a dummy list of books for the tests. Use different data so you can tell that the right list is being used:

```
// A test list of books
const testBooks: Book[] = [{
    title: 'The Hobbit',
    author: 'J R R Tolkien',
    cover: '',
    bookId: 1
}, {
    title: 'A Wizard of Earthsea',
    author: 'Ursula K Le Guin',
    cover: '',
    bookId: 2
}];
```

   b. Just after the dummy list, create a fake `BookService` and have it return the dummy data:

```
// Create a fake BookService object
let bookService: any = jasmine.createSpyObj('BookService',
    ['getBooks']);
bookService.getBooks.and.returnValue(of(testBooks));
```

   c. Use the fake `BookService` in the testing module:

```
TestBed.configureTestingModule({
    declarations: [
        BookPageComponent,
        MockBookListComponent,
        MockBookFormComponent
    ],
    providers: [
        { provide: BookService, useValue: bookService }
    ]
})
    .compileComponents();
```

d. Change your test to expect the new data.

e. While we were getting the data in the component, there was no reason to test that the array contained the right books. Now there is. Write a test to check that the component has retrieved the data from the service. A simple test will suffice. (Your test won't pass until you complete the next step.)

```
it('should retrieve books from the service', () => {
    expect(component.books.length).toBe(2);
    expect(component.books[0].title).toBe('The Hobbit');
    expect(component.books[1].title)
        .toBe('A Wizard of Earthsea');
});
```

f. *Reminder:* your `MockBookFormComponent` must have an `@Output` property and it must define an `add()` method.

10. Open `book-page.component.ts` and change it to use the `BookService`.

a. Add or modify the class constructor, injecting a `private bookService` property of type `BookService`.

b. Create a `getBooks()` method that calls `bookService.getBooks()` and `subscribe()` assigns the return to `this.books`.

c. Call the `getBooks()` method inside the `ngOnInit()` method.

d. Replace the fixed initialization of `books` with an initialization to an empty array.

Your code will look something like this:

```
constructor(private bookService: BookService) { }

ngOnInit() {
   this.getBooks();
}

getBooks() {
   this.bookService.getBooks()
       .subscribe(data => this.books = data);
}
```

11. All of your tests should pass except for one in `book-page.component.spec.ts` that adds a book. You'll fix that next.

12. Let's switch our attention to the `addBook()` method of `BookPageComponent`.
    a. Delete existing test(s) for `addBook` because they check that the array is being maintained correctly and that is now the responsibility of the service. Instead, we need a test to ensure that the service is being called correctly.
    b. Amend the `BookService` spy to mock `addBook` as well as `getBooks`.
    c. Declare a second spy `addBookSpy` to track calls to the service's `addBook()` method. In order for `addBookSpy` to be visible to the specs, it must be declared in *inside* `describe()` but *before* `beforeEach()`:

```
describe(() => {
    let addBookSpy: any;
    beforeEach(waitForAsync(() => {
        …
        addBookSpy = jasmine.createSpyObj(…);
```

    d. Initialize `addBookSpy` *inside* `beforeEach()`. A call to the fake `addBook()` method should return its parameter as an Observable.

```
addBookSpy = mockBookService.addBook.and.callFake((param: any) => {
   return of(param); // return an Observable that wraps the param
});
```

    e. Modify the existing `addBook()` method in `BookPageComponent` so that it calls the `bookService.addBook()` method and calls `getBooks()` in the subscribe. It does not need the data returned by `addBook()`, but by adding the `getBooks()` in the subscribe, we can guarantee that the `addBook()` call has completed before the `getBooks()` is executed.
    f. Your `addBook()` method will look something like this:

```
addBook(book: Book) {
    this.bookService.addBook(book)
        .subscribe(() => this.getBooks());
}
```

g. The test(s) should now pass. They may look something like this (you will only have the second test if you did the bonus item from Exercise 8.2):

```
it('should call the service to add a book', () => {
    const expected = new Book('The Lathe of Heaven',
        'Ursula K Le Guin', '', 3);
    component.addBook(expected);
    expect(addBookSpy).toHaveBeenCalledWith(expected);
});

it('should respond to output event from book form', () => {
    const expected = new Book('The Silmarillion',
        'J R R Tolkien', '', 3);
    // Get the mock book form component
    const bookForm = fixture.debugElement.query(
        By.css('app-book-form')).componentInstance;
    // Set the book
    bookForm.book = expected;
    // Trigger the output event
    bookForm.add();
    // Now check the method was called
    expect(addBookSpy).toHaveBeenCalledWith(expected);
});
```

13. Verify that everything is still working, and all your tests are passing.
    a. Well done!
14. Open `book-page.component.spec.ts` and review the `describe()` call after the TODO comment.
    a. The specs in this `describe()` demonstrate how to configure a mock `BookService` that allows us to define different test scenarios for the `BookPageComponent`.

Exercise 9.2: Retrieving and Adding Data with REST

Time: 45 minutes

In this exercise, you will retrieve book data from a RESTful web service and display it inside your application. You will convert the existing book service (from the previous exercise) to communicate with a RESTful web service that is running on the Tomcat server. To do this, you will use the Angular Http service.

Note: The `book.service.spec.ts` file in the solution project is slightly different but will produce similar results.

Deploy the BookService on Tomcat

1. Your instructor will inform you of the location of Java RESTful BookService.
    a. This is available as a war file (`BookService.war`).
    b. This is a web archive that can be used directly in Tomcat.
2. To run `BookService.war`:
    a. Copy the file to Tomcat's `C:\Apps\tomcat\webapps` folder.
    b. Start the Tomcat server by double-clicking **`C:\Apps\tomcat\bin\startup.bat`**.
    c. Do ***not*** close Tomcat's command window.
3. Verify the BookService has deployed correctly by visiting the following URLs in your browser:
    a. `http://localhost:8080/BookService/`
    b. `http://localhost:8080/BookService/jaxrs/books`

Build the Angular Application

4. We will convert the BookService to use http. But let's start by changing the `getBooks()` test to use the `HttpClientTestingModule`.

   a. Edit `book.service.spec.ts` and set up the `HttpClientTestingModule` by replacing the call to `beforeEach()` with the version shown below:

```
let httpTestingController: HttpTestingController;

beforeEach(() => {
    TestBed.configureTestingModule({
        imports: [
            HttpClientTestingModule
        ]
    });
    httpTestingController = TestBed.inject(HttpTestingController);
});
```

   Note: `beforeEach` does not initialize the service, so the 'should be created' spec will fail. Feel free to delete that spec since it is not important.

   b. The new imports are from `@angular/common/http/testing`.

   c. The `getBooks()` method will make a single `GET` to the RESTful service.

   d. Change the get test so it expects a single call to the appropriate URL and returns a set of test books. For the test books, you can copy the list of books from the BookService (we'll delete it from BookService in a later step.)

```
it('should return books', inject([BookService],
        fakeAsync((service: BookService) => {
    let books: Book[] = [];
    service.getBooks()
        .subscribe(data => books = data);
    const req = httpTestingController.expectOne(
        'http://localhost:8080/BookService/jaxrs/books');
    // Assert that the request is a GET.
    expect(req.request.method).toEqual('GET');
    // Respond with mock data, causing Observable to resolve.
    req.flush(testBooks);
    // Cause all Observables to complete and check the results
    tick();
    expect(books[0].title).toBe('The Lord of the Rings');
})));
```

   e. This will fail because the service is not yet using http.

5. Now, let's update the service to use http.

6. Open `app.module.ts` and add `HttpClientModule` to the `imports` array. Accept the suggestion to add a file `import` from `@angular/common/http` or do it manually. Make especially sure it does not come from Selenium.
7. Add `HttpClient` to `book.service.ts`.
    a. Use a `constructor` to inject a `private HttpClient` property `http` into the `class`.
    b. Make sure the file `import` for `HttpClient` comes from `@angular/common/http`.
8. Add a `public` property `url` to the class, with the value `http://localhost:8080/BookService/jaxrs/books`.
9. Delete the current code inside the `getBooks()` method.
    a. Replace it with code returning the result of a call to `this.http.get()`.
    b. Parameterize the `get` with `Book[]`.
    c. Pass the base `url` as a single argument to `get()`.

```
getBooks(): Observable<Book[]> {
    return this.http.get<Book[]>(this.url);
}
```

10. Run your application and your tests.
    a. The application should display a list of books from the service.
    b. The Add Book functionality will not work.
    c. The test for `getBooks()` should pass. The test for `addBook()` will fail because we have not addressed it yet.
11. We will now fix the `addBook()` test and functionality.

12. First, change the test so it uses the `HttpClientTestingModule` to check that a `POST` has been sent. We are not interested in the return value any more since that is the responsibility of the remote service.

```
it('should POST to add a book', inject([BookService],
  fakeAsync((service: BookService) => {
    const expected = new Book('A Wizard of EarthSea',
        'Ursula K Le Guin', '', 3);
    service.addBook(expected)
        .subscribe();
    const req = httpTestingController.expectOne(
        'http://localhost:8080/BookService/jaxrs/books');
    // Assert that the request is a POST.
    expect(req.request.method).toEqual('POST');
    // Assert that it was called with the right data
    expect(req.request.body).toBe(expected);
})));
```

13. Now, change `addBook()` to use the `HttpClient` service.

    a. Delete all the content from the existing `addBook()` method and replace it with new code declaring a variable `headers` and assigning a `new HttpHeaders()` object as the value.

    b. Pass an object literal to the `HttpHeaders()` constructor. The object literal should have a single string literal property `Content-type`, with the string literal value `application/json`.

    c. Have it call the `http post` method passing in the `url`, the book to add, and the headers.

```
addBook(book: Book): Observable<Book> {
    const headers = new HttpHeaders({
            'Content-type': 'application/json'
    });
    return this.http.post<Book>(this.url,
        book, { headers: headers });
}
```

14. You can now delete the local mock data `books` from the `BookService`, as it is no longer needed.

    Make sure everything is working and all your tests pass.

Bonus Exercise (to be attempted if time permits)

15. Edit `book-service.spec.ts` and review the `describe()` call at the end of the file.
    a. `BookService` isn't an Angular component, so it doesn't require Angular testing support. It can be tested without calls to the Angular functions `inject()`, `fakeAsync()`, or `tick()`.
    b. The specs in this `describe()` demonstrate this alternate approach to testing Angular services.
    c. For more information about testing Angular services, visit https://angular.io/guide/testing-services.

Exercise 9.3: Handling Errors in a RESTful Service

Time: 20 minutes

In this exercise, you will create an error handler for your http service and test it.

1. Verify that Tomcat is running. If necessary, run `C:\Apps\tomcat\bin\startup.bat`.
2. Open the `BookService`.
3. Create the error handler:

```
handleError(response: HttpErrorResponse) {
  if (response.error instanceof ProgressEvent) {
   console.error('A client-side or network error occurred; ' +
   `${response.message} ${response.status} ${response.statusText}`);
  } else {
    console.error(`Backend returned code ${response.status}, ` +
     `body was: ${JSON.stringify(response.error)}`);
  }
  return throwError(
    () => 'Unable to contact service; please try again later.');
}
```

4. Use the error handler from the `getBooks()` method:

```
import { catchError } from 'rxjs/operators';
…
getBooks(): Observable<Book[]> {
    return this.http.get<Book[]>(this.url)
        .pipe(catchError(this.handleError));
}
```

5. Now write a test for a 404 error. It might look like this:

```
it('should handle a 404 error', inject([BookService],
        fakeAsync((service: BookService) => {
    let errorResp: HttpErrorResponse;
    let errorReply: string = '';
    const errorHandlerSpy = spyOn(service,'handleError')
                                .and.callThrough();
    service.getBooks()
        .subscribe({next: () => fail('Should not succeed'),
                    error: (err) => errorReply = err});
    const req = httpTestingController.expectOne(service.url);
    // Assert that the request is a GET.
    expect(req.request.method).toEqual('GET');
    // Respond with error
    req.flush('Forced 404', {
        status: 404,
        statusText: 'Not Found'
    });
    // Cause all Observables to complete and check the results
    tick();
    expect(errorReply).toBe(
        'Unable to contact service; please try again later.');
    expect(errorHandlerSpy).toHaveBeenCalled();
    errorResp = errorHandlerSpy.calls.argsFor(0)[0];
    expect(errorResp.status).toBe(404);
})));
```

6. Check that everything works.
7. Now make the `BookPageComponent` display an error message.
    a. Declare a variable in `BookPageComponent`:

    ```
    errorMessage: string = "";
    ```

    b. Add a `DIV` in the `BookPageComponent` html template to display it.

    ```
    <div *ngIf="errorMessage" class="error">
        {{errorMessage}}
    </div>
    ```

c. Add handling to the subscribe in `BookPageComponent`.

```
getBooks() {
    this.bookService.getBooks()
        .subscribe({
                next : (data) => {
                    this.books = data;
                    this.errorMessage = '';},
                error: (e) => this.errorMessage = e });}
```

8. How could you write a test for this functionality in `BookPageComponent`? Perhaps something like this:

```
it('should display an error message', () => {
    let errorDiv = fixture.debugElement.nativeElement
        .querySelector('.error');
    expect(errorDiv).toBeFalsy();
    component.errorMessage = 'An error';
    fixture.detectChanges();
    errorDiv = fixture.debugElement.nativeElement
        .querySelector('.error');
    expect(errorDiv).toBeTruthy();
});
```

9. How can you simulate a failure to check this for yourself?
   a. Try stopping the service and reloading the page.
   b. Or changing the URL in the service.

Bonus Exercise (to be attempted if time permits)

10. Write a new test for a network failure.
11. Style the `DIV` appropriately.

This page intentionally left blank.

Chapter 10: Angular End-to-End (E2E) Testing Applications

Exercise 10.1: Designing E2E Testing

Time: 20 minutes

In this exercise, you will design two E2E test scenarios for the BookStore application. You may consider any scenarios you wish, however in the following exercises, we will use:

1. A user adds a book to the list.
2. There is a new requirement to add a navigation bar to the application. The navigation bar contains a link with the text "About", which navigates to a page with details about the application (version number, contact information, etc.)

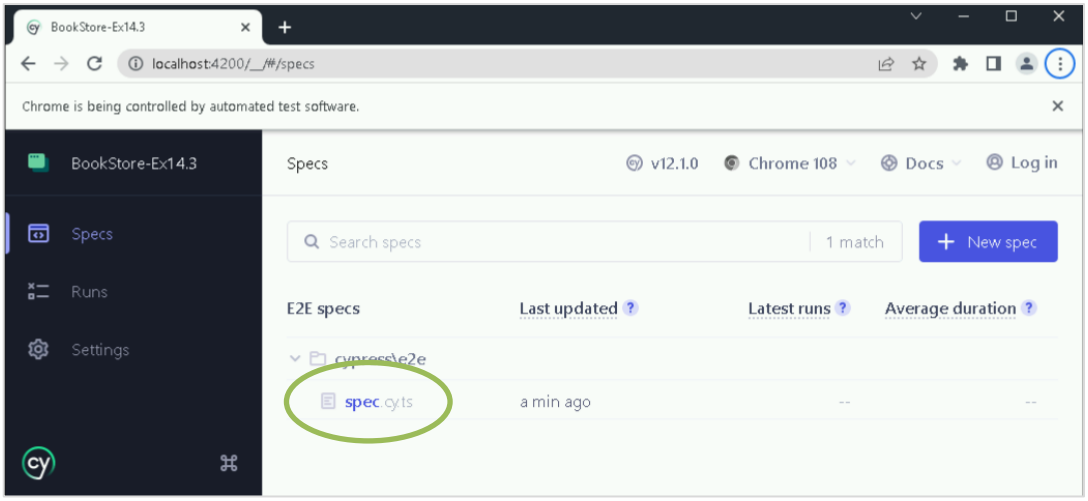
Remember to identify user functions, conditions, and test cases.

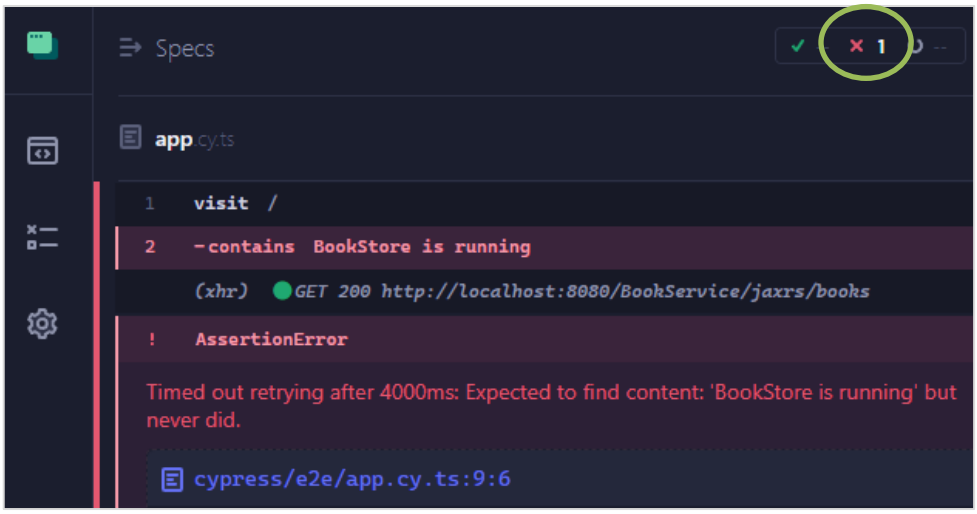
Exercise 10.2: Writing a Simple Cypress Test

Time: 30 minutes

In this exercise, you will write an E2E test using Cypress. The test will verify the contents of the table of books.

1. Confirm that the backend BookService is deployed to Tomcat by browsing to http://localhost:8080/BookService/jaxrs/books
   a. If necessary, start Tomcat with `C:\Apps\tomcat\bin\startup.bat`.
2. Run E2E tests using Cypress.
   a. Launch the Cypress UI:
      `ng e2e`
   b. If Cypress displays a "What's New in Cypress" popup, click **Continue**.
   c. Under **Choose a Browser**, note that Chrome is selected by default. Click the **Start E2E Testing in Chrome** button.
   d. Run the generated `spec.cy.ts` E2E test by clicking the link in the UI (the spec will fail because you changed the welcome message on the home page).





3. Change the spec file name to `app.cy.ts` (your spec file is in `BookStore/cypress/e2e`). Correct the spec and save your changes. Click the new file name in the E2E specs list, and Cypress will run it again. Verify that it passes.
4. Now open the BookStore application in Chrome and go through the steps you expect the test to take.
    a. Navigate to "/" (the home page).
    b. How do you know the content is correct? What do you look for?
        i. The "Welcome" header
        ii. The "Books" header
        iii. A table with 4 rows.
        iv. Column 1 of the first table row has the value "Design Patterns"
        v. Column 1 of the last table row has the value "Cryptonomicon"
5. Now add a new spec to `app.cy.ts`. The spec will perform the same checks that you did in your manual test above:
    a. Navigate to "/".
    b. Verify there are <h1> and <h2> headers with the correct content.
    c. Verify that the table has 4 rows.
    d. Verify column 1 of the first and last table rows.

e. Your code may look something like this; be sure to change the CSS selectors as needed to match your page's HTML structure:

```
it('contains headers and a table of books', () => {
  cy.visit('/');
  cy.get('app-root h1').contains('Welcome to Angles on Books!');
  cy.get('app-root h2').contains('Books');

  cy.get('tbody > tr').should('have.length', 4);
  cy.get('tbody > tr:nth-child(1) > :nth-child(1)')
    .invoke('text').should('eq', 'Design Patterns');
  cy.get('tbody > tr:last-child > :nth-child(1)')
    .invoke('text').should('eq', 'Cryptonomicon');
});
```

6. Create the two page objects to support this spec.

a. The app page object (`app.po.ts`) may look like this (change the CSS selectors to match your application). Create this file in the `<project>/cypress/support` folder:

```
export class AppPageObject {
  navigateToHomePage() {
    cy.visit("/");
  }
  checkTitle(title: string) {
    cy.get('app-root h1').should('contain.text', title);
  }
}
```

b. The table page object (`table.po.ts`) may look like this (change CSS selectors as necessary):

```
export class TablePageObject {
  checkBookTableHeader(tableHeader: string) {
    cy.get('app-root h2').should('contain.text', tableHeader);
  }
  getBookTableRows() {
    return cy.get('tbody > tr');
  }
  getBookTableFirstRowTitleColumn() {
    return cy.get('tbody > tr:nth-child(1) > :nth-child(1)')
             .invoke('text');
  }
  getBookTableLastRowTitleColumn() {
    return cy.get('tbody > tr:last-child > :nth-child(1)')
             .invoke('text');
  }
}
```

c. The spec in `app.cy.ts` will look something like this (remember to import the page object classes):

```
let appPage: AppPageObject = new AppPageObject();
let tablePage : TablePageObject = new TablePageObject();
```

d. Write tests using these page objects and their methods.

7. Run the tests.
   a. Work on them until they pass.
8. When you are satisfied with your specs, close the Cypress UI.
   a. Then press CTRL+C in the command window where you ran `ng e2e`.

Bonus Exercises (to be attempted if time permits)

9. Make your `TablePageObject` class more reusable by adding arguments to methods where appropriate. Then add another spec in `app.cy.ts` that calls your new methods. For example:

```
tablePage.getBookTableRowColumn(3, 1).should('eq', 'Clean Code');
tablePage.getBookTableRowColumn(3, 2).should('eq', 'Robert Martin');
```

10. Add a method to your `TablePageObject` that returns the value of a given column from the first row. Add another method that returns a given column's value from the last row. Both methods should take an argument that is the number of the desired column.
    a. In `app.cy.ts`, add a new spec that verifies the values of the title and author columns in the first and last rows. Make sure your new spec passes.

Exercise 10.3: E2E Tests that Enter Data in HTML Inputs

Time: 30 minutes

In this exercise, you will write a Cypress E2E test that adds a new book.

1. Start by opening the application and going through the steps you expect the test to take.
    a. Enter a book title and author.
    b. Click the button.
    c. How will you identify the input controls and the button? How will you know if the right book was added to the list?
2. Write the test code in a new spec file, `book.cy.ts`. Your new test should perform the following actions:
    a. Navigate to the app page.
    b. Use a book page object for all interactions with the book page.
    c. Check that your chosen book title is not already on the page.
    d. Use a book page object to get a reference to the input controls and enter text.
    e. Also use the book page object to get a reference to the Add Book button and click it.
    f. Look for the right data on the page.
    g. Your code may look a little like this, but the details may differ:

```
it('should add a book', () => {
     const title = 'Zero History';
     app.navigateToHomePage();

     cy.contains(title).should('not.exist');
     book.checkAddButtonDisabled();

     book.addAuthor('William Gibson');
     book.addTitle(title);

     book.checkAddButtonEnabled();
     book.clickAddBook();
     cy.contains(title).should('exist');
 });
```

h. And your page object may look like this:

```
export class BookPage {
    addTitle(title:string) {
        cy.get('app-book-form input#title').type(title);
    }
    addAuthor(name:string) {
        cy.get('app-book-form input#author').type(name);
    }
    clickAddBook() {
        cy.get('button').contains('Add Book').click();
    }
    checkAddButtonDisabled() {
        cy.get('button').contains('Add Book').should('be.disabled');
    }
    checkAddButtonEnabled() {
        cy.get('button').contains('Add Book').should('not.be.disabled');
    }
}
```

3. If your test doesn't work the first time, you may need to stop and restart the service to re-run the test, otherwise the book will still be present.
    a. This is inconvenient and compromises our ability to use this test in an automated build process. Let's change the test so that it doesn't use a fixed value for the title.
    b. There are libraries available that generate random "realistic" string data, but for our purposes, we will use `uuid`, which is already installed.
    c. Add an import to the spec file: `import { v4 as uuid } from 'uuid';`
    d. You may now receive a compiler error: 'uuid module has no type declarations file'.
        i. Fix this by accepting VS Code's quick fix to install `@types/uuid`.
    e. And replace the static title with a call to `uuid()`:

```
const title = uuid();
```

    f. The series of book titles is rubbish, but the test is now repeatable.

Bonus Exercise (to be attempted if time permits):

4. While the page object is acceptable in the current test, there is usually some advantage in encapsulating more functionality since many operations (like adding a new book) will be used over and over again in a "real-life" test.

   a. Add an `addBook()` method to the book page object:

```
addBook(title:string, author:string){
    cy.get('app-book-form input#title').type(title);
    cy.get('app-book-form input#author').type(author);
    cy.get('button').contains('Add Book').should('not.be.disabled');
    cy.get('button').contains('Add Book').click();
}
```

   b. Change the spec to use this method.

```
book.addBook(title, 'William Gibson');
```

Chapter 11: Building an Application

Exercise 11.1: Building an Angular Application

Time: 180 minutes

In this exercise, you will build a new Angular application and view it in the Google Chrome browser. The application will display information about cars.

For simplicity, we will put all the code in the root module.

The best way to write this application is to start with the service. By doing this, we will know exactly what the service interface looks like, making it easy to mock. However, that would not show any visual feedback. At this stage of experience, there is value in seeing whether the application is working, so we will start with the list of cars and work from there.

At each stage of your project, the tests should work, and you should create tests for every part of the application.

Your instructor will tell you where to find the Cars service and what URL it uses. Copy the `CarService.war` file to Tomcat's webapps folder (`C:\Apps\tomcat\webapps`). If Tomcat is not running, start it by running `C:\Apps\tomcat\bin\startup.bat`.

Verify the service is running by opening `http://localhost:8080/CarService` in any browser. Click the links to view the data and see the full URLs. The route to the list of Cars is `http://localhost:8080/CarService/jaxrs/cars`.

Build the Application

1. Open the folder `D:\WebAppsClient\Ch11\ManageCars` in VS Code and examine the directory and file structure.
    a. This is a template Angular project that was created with `ng new`.
2. From the VS Code menu bar, select **Terminal > New Terminal**.
    a. Verify the current directory is `D:\WebAppsClient\Ch11\ManageCars`.
3. Build and run the application by typing the following commands in the terminal window:

```
npm install

npm start
```

4. Open another terminal window and run the tests. Make sure that your tests pass as you continue to work on your application.
5. Open Google Chrome and navigate to `http://localhost:4200`.
6. First, we will make a minor cosmetic change by modifying the title property to confirm everything is working.
    a. In true TDD style, modify the tests to expect the new title value of '`Cars World`'. Notice that you now have a failing test. Fix failing test.
    b. Open `app.component.ts` in the `src/app` folder, and change the title to be `Cars World`.
    c. Open `app.component.html`, delete all lines, and then add HTML semantic tags to define the basic structure of the page.
    d. Add a line with a header:

    ```
    <h1>{{ title }} app is running!</h1>
    ```

    e. Add a copyright notice with the current date centered at the bottom of the page.
    f. Define a few CSS styles and use them in your template code.
    g. Check that it displays as expected.
7. Fix the tests if necessary.
    a. Add or remove tests as needed.

The Car Class

8. Inside the `app` folder, create a `models` folder. Inside the newly created folder, create a file called `car.ts` and add a class with a constructor to reflect the Cars data structure that is returned by the web service. You can use `ng generate class models/car` (make sure your terminal is in the correct folder before executing commands).

    Hint: Looking at the output from the RESTful service should give you some insight on how to create the `cars.ts` class
9. Do all your tests pass?
    a. If not, get your tests to pass before moving forward.

    Hint: You may need to stop the tests (CTRL+C) and then start them again (`npm test`)

    b. It is highly recommended that you fix the unit tests before moving on to the next step.

Make the CarList Component

10. Make a new component and call it `CarList`.

    ```
    ng generate component CarList
    ```

11. Add it to the app component template and check that the placeholder text appears.
12. Fix the app component test by defining a mock `CarList` component.
13. Create a cars list as a property.
    a. Declare the property.
    b. Initialize the property to a fixed array of Cars. You should have two cars in the list to ensure the code handles the array properly.
14. Next, change the `car-list-component.html`.
    a. Replace everything with HTML elements of your choosing.
    b. Include a table and a table row with *`ngFor` and several `<td>` tags with car properties using interpolation binding.
    c. Arrange the properties in an order that makes sense to you (do not just use the default order in the class).
15. At this point, verify that the cars have been added to your running application.
16. Write a simple test that looks for the table and checks it has some data in it.

Create the Cars Service

17. From the `ManageCars` folder, create a service module by running the following statement:

    ```
    ng generate service cars/car
    ```

18. Open the `car.service.ts` file.
    a. Create a `getCars()` method. This should return `Observable<Car[]>`.
    b. Create a `mockCars const` outside of the `getCars` function and initialize it. If you initialize it to different values, you will be able to tell that the correct data is shown.
    c. Return `mockCars` using `of()`.
19. Change the `car-list.component.ts` file to use `CarService`.
    a. Add a private attribute named `carService` of type `CarService` to the constructor. This will automatically trigger adding an import for `CarService` to your file.
    b. Define the `loadAllCars` method to call the service's `getCars()` to retrieve the cars from backend API.
    c. Change `ngOnInit()` to call `this.loadAllCars();`
    d. You can remove the initialization of `cars` from `CarListComponent`.
20. Make sure everything works so far.
21. Update the tests for the `CarListComponent`, so they use a mock `CarService`.
    a. You must define an array of cars that can be returned by the mock service method.

Convert CarService to Use HTTP

22. Set up the `HttpClient` service.
    a. Add `HttpClientModule` to the imports array of `AppModule`. Make sure it adds a file import from `@angular/common/http`.
    b. Use a `constructor` to inject a `private http` of type `HttpClient` into the `CarService class`. Let VS Code fix the missing import for you, but make sure to select `HttpClient` from `@angular/common/http` rather than selenium. Lots of tests might break at this point. We will fix them.
23. Add a private property in the service class for the base URL of the backend API, with the value `http://localhost:8080/CarService/jaxrs/cars?filter=`

24. Amend the service's `getCars()` method.
    a. The method should now return the results of a call to `this.http.get()`.
    b. Parameterize the `get` with `Car[]`.
    c. Pass the base URL as a single argument to `get()`.
    d. Remove the `mockCars` from `CarService`.
25. Check that your application now displays a list of cars from the web service.
26. Fix the tests for the `CarService`.
    a. The test should use `HttpClientTestingModule`.
    b. You will need a mock array of `Cars`.
    c. For now, it will be enough to test that the service makes a call to the right URL.

Add the Buttons

27. Now, create an additional method in the service.
    a. Open the `CarService`.
    b. Add a `getCarsByPrice` function like `getCars()`, but pass a URL that appends `price` to the base URL. (This URL gets a list of the top three cars by price).
28. Add similar methods to `car-list.component.ts` that delegate to the service.
29. Add two buttons to `car-list.component.html`: one for loading all cars, one for cars sorted by price. Put the buttons in a `nav` element.
30. Bind click events to those buttons calling `loadAllCars()` and `loadCarsByPrice()` respectively, e.g., `(click)="loadAllCars()"`.
31. Verify that the buttons work as planned. Note that there should only be one table in the template, but the contents should vary depending on which button you have pressed.
32. Add two more buttons:
    a. A button that loads the top 3 cars by the model year (append `year` to the base URL).
    b. A button that loads the top 3 cars by the number of doors (append `doors` to the base URL).
33. Bind the new buttons' click events to new methods in the car list component. Verify all buttons work.

Refactor Your Classes to Remove Duplicate Code

34. Do the methods of your car list component contain duplicate code? Look for refactoring opportunities to extract duplicate code into private helper methods.
    a. Note that you may need to perform a similar refactoring in your service class to eliminate duplicate code in your car list component.
    b. For example, you can create a service method `getCarsWithFilter()` that takes a string parameter `filter` and returns `Observable<Car[]>`. Then, you can modify all the car list component methods to call the service's `getCarsWithFilter()`, passing an appropriate value as the filter.

Write Automated E2E Tests

35. Add E2E specs in the `cypress/e2e` folder. Be sure to test all success paths through the application.
36. Simplify your E2E specs by refactoring complex operations into page object classes. Create your page object class definitions in the `cypress/support` folder.
37. Run `ng e2e` and verify that all E2E specs pass.

Bonus Exercise (to be attempted if time permits)

38. Style the application.
39. Add unit tests for all the functionality, including at least one negative test for the HTTP service.
40. Look for refactoring opportunities.

Chapter 12: Pipes

Exercise 12.1: Creating a Custom Pipe

Time: 30 minutes

`Book` objects have a `cover` property. Currently, all these are empty. The data returned from the RESTful web service should be a mix of image URLs and empty strings. You will create a pipe that displays a default image when no cover image URL is provided.

1. We will put the pipe in a new module, which we will call `SharedModule`.
    a. Run `ng g module shared`.
2. Create the `NoImagePipe`.
    a. Run `ng g pipe --module shared shared/no-image`.
    b. Add an `exports` array to the module and add `NoImagePipe` to it.

```
@NgModule({
    declarations: [
        NoImagePipe
    ],
    imports: [
        CommonModule
    ],
    exports: [
        NoImagePipe
    ]
})
export class SharedModule { }
```

3. This may break your tests.
    a. You may need to stop the tests (CTRL+C in the test terminal), then restart the tests. They should now be in sync with your latest changes.

4. Create a unit test for the `NoImagePipe`. When the image is supplied (parameter to the transform method), it should be passed through.

```
it('should pass a specified image through', () => {
    const pipe = new NoImagePipe();
    expect(pipe.transform('a.png')).toEqual('a.png');
});
```

Note: This is an example of a TDD or Test-Driven Development: you write a test (i.e., a specification) of the `transform` method *before* you implement it, so the test will initially fail; and after you implement the method correctly, the test will pass. Remember the TDD mantra: Red, Green, Refactor.

5. To make this test pass, you must complete the implementation of the `transform()` method in the `NoImagePipe` class. Open `no-image.pipe.ts` and make the following modifications to `transform()`:
    a. Change the data type of the `value` parameter to `string`.
    b. Delete the unused second parameter.
    c. Change the return type of the method to `string`.
    d. Instead of returning `null`, return the `value` parameter.
    e. Verify the test now passes.
6. Add another test so that when there is no image supplied (parameter to the transform method is empty string), the value returned should be `/assets/images/NoImage.svg`.

```
it('should use default if there is no value', () => {
    const pipe = new NoImagePipe();
    expect(pipe.transform(''))
        .toEqual('/assets/images/NoImage.svg');
});
```

Note: Another Excellent TDD test. It tests one unit of work only.

7. Implement the code to make that test pass.

```
transform(value: string): string {
    return value ? value : '/assets/images/NoImage.svg';
}
```

8. Find this image in the `Ch09` folder and put it in the appropriate folder.
    a. In VS Code, you will need to create an `images` folder and place the `NoImage.svg` in that folder.
9. Add an import for `SharedModule` to the `BooksModule`.
    a. By now, you should know how to add the `SharedModule` to the imports array.
    b. You may also have noticed that if you add it to the imports array, Visual Studio Code may help you by adding the import statement at the top of the file.
10. Open `book-list.component.html` and add an additional column to the `table`. (You will need to add both a `th` and a matching `td`.)
    a. The `td` should contain an `img` element data bound to `book.cover`.
    b. Use a style to restrict it to 100px in size (use `max-width` and `max-height`).
    c. Use an additional binding for the `alt` attribute, binding it to the `title` of the `book` concatenated with a space and the string literal `book cover`.
    d. Apply the `noImage` pipe to the `book.cover` binding in the `src` attribute.

```
<td><img src="{{ book.cover | noImage }}"
    alt="{{book.title + ' book cover'}}" /></td>
```

11. View your application again by browsing to `localhost:4200`.
    a. This time, you should see the default image.
    b. But your unit tests will fail; you'll fix that in the next step.
12. Your tests no longer work because `BookListComponent` now also depends on `NoImagePipe`.
    a. Fix the dependencies.
    b. The way that guarantees this will not be a problem in the future, is to create a mock pipe in `book-list.component.spec.ts`.

```
@Pipe({
    name: 'noImage'
})
class MockNoImagePipe implements PipeTransform {
    transform(value: string): string {
        return value;
    }
}
```

This page intentionally left blank.

Chapter 13: Angular Routing

Exercise 13.1: Routing with the Angular Router

Time: 20 minutes

In this exercise, you will add routing to your Single Page Application. Before adding routing to the application, you will need to create a component that will act as the endpoint of the route. For this, you will create an About page.

1. Switch back to the BookStore application.
2. Use the CLI to create a new component named `about-page`.

   Hint: `ng g component about-page`

   a. This is a "page" component since it will have its own URL.

   b. Inside `about-page.component.html`, use semantic tags and CSS styles to describe your application.

      Hint: A footer that is centered is a good place to start.
3. Create the `AppRoutingModule`.

   a. Using the CLI, create a new module named `app-routing`. Put it in the root of the project rather than in its own directory since that is where it would have been put if we had allowed the CLI to create it originally:

      ```
      ng g module --flat=true app-routing
      ```

   b. Declare a `const routes` of type `Routes` and set it equal to an array. Add two object literals to the array. The first should define the empty `path` and should use the `component BookPageComponent`. The second should have a `path about` and should use the `AboutPageComponent`.

      Hint: See the course notes for an example (or do a web search as needed).

   c. Add the argument `RouterModule.forRoot(routes)` to the imports array.

   d. Add any missing file imports (`Routes` and `RouterModule` are from `@angular/router`).

   e. Add an `exports` property set to an array with the single argument `RouterModule`.

   f. There are no declarations and the `CommonModule` is not needed.

g. Your module may look like this:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AboutPageComponent } from
        './about-page/about-page.component';
import { BookPageComponent } from
        './books/book-page/book-page.component';

const routes: Routes = [{
    path: '',
    component: BookPageComponent
}, {
    path: 'about',
    component: AboutPageComponent
}];

@NgModule({
    imports: [RouterModule.forRoot(routes)],
    exports: [RouterModule]
})
export class AppRoutingModule { }
```

4. Modify the `app.module.ts` file by doing the following:
    a. Add `AppRoutingModule` to the module imports.
    b. Check that `AboutPageComponent` is a module declaration.
5. Open `app.component.html` and delete the `<app-book-page>` element from the html template. Replace it with `<router-outlet>`.
6. Verify that the application runs as before.
    a. Add `/about` at the end of `localhost:4200` in the browser's address bar and click the Enter key. You should be taken to your new about page.
    b. Use the browser's back button to be taken back to the home page.
7. You just broke all the tests for `AppComponent` because it now relies on `<router-outlet>`, which is not defined.
    a. The easiest way to fix this is to add `RouterTestingModule` to the imports of the `TestBed` in the `AppComponent` tests.

8. Modify the `template` in `app.component.html` by adding two HTML `a` tags at the top of the page.
    a. Bind the `a` tags `routerLink` attributes to the strings `/` and `/about`, and set the text to `home` and `about`, respectively.
    b. Wrap the links in a `nav` element.
9. Verify the router links work correctly.

Bonus Exercise (to be attempted if time permits)

10. Now you will write Cypress E2E tests for the new About page.
11. First, open the application and go through the steps you expect the test to take.
    a. Click the **About** link.
    b. Inspect the About page, looking for items that will identify whether you are on the right page.
12. Write the test code in a new spec file `about.cy.ts`. Add code to make Cypress perform the following steps:
    a. Navigate to the app page.
    b. Use an app page object to click the About link.
    c. Use an about page object to get some item that identifies your page.
    d. Your code may look a little like this (change CSS selectors as needed):

```
it('should navigate to about page', () => {
  cy.visit('/');
  // get the link to the About page and click it
  cy.get('app-root nav a').contains('About').click();
  //check the content of the about page
  cy.get('app-about-page p')
    .should('contain', 'This is the Angles on Books application');
});
```

    e. Use Cypress to run your new spec. Verify that it passes.

13. Now simplify the spec by using page objects.
    a. Add a new method to the about page object in `app.po.ts` to click the About link on the home page:

```
clickAboutLink() {
  cy.get('app-root nav a').contains('About').click();
}
```

    b. Create an about page object in `about.po.ts`. It may look like this:

```
export default class AboutPage {
  getAboutContent() {
    return cy.get('app-about-page p');
  }
}
```

    c. The setup code for the spec in `about.cy.ts` will look a little like this. You will also need to import the support pages:

```
let app: AppPage = new AppPage();
let about: AboutPage = new AboutPage();
```

    d. Write tests using these page objects and their methods.
14. Run the tests.
    a. Work on them until they pass.

Exercise 13.2: Passing and Receiving Route Parameters

Time: 30 minutes

In this exercise, you will pass parameters as part of a route and retrieve the values inside the destination component. Your goal in this exercise is to set up parameterized routing to a new component in a new module. The new component will display book book IDs. In a later exercise, you'll add a page that displays book reviews.

1. Create a new module for reviews.
    a. Using the CLI, create a new `reviews` module using the `--routing` option.
2. Create a `ReviewPageComponent`.
    a. Using the CLI, create a new `review-page` component in the `reviews` folder and add it to the `reviews` module (putting it in the right folder will automatically add it to the module):

    ```
    ng generate component reviews/review-page
    ```

    b. Declare the `bookId` property and set its value to `-1`.
    c. Amend `review-page.component.html`, so it contains a binding to write the variable `bookId` into the page.
3. Define the `ReviewsRoutingModule`. In `reviews-routing.module.ts`, in the routes array, declare a path property `reviews/:id` and a component `ReviewPageComponent`:

    ```
    const routes: Routes = [{
        path: 'reviews/:id',
        component: ReviewPageComponent
    }];
    ```

4. Add `ReviewsModule` to the array of imports in `app.module.ts`.
5. Add `RouterModule` to the imports array of `books.module.ts`.
6. In `book-list.component.html`, add a `routerlink` around `book.title`:

    ```
    <td>
      <a [routerLink]="['/reviews', book.bookId]">{{book.title}}</a>
    </td>
    ```

7. Test your application in the browser. You should be able to access your `reviews-list` component by clicking any book title on the home page.
    a. You can access the `ReviewPageComponent`, but the book ID is always -1 because the component doesn't access the route parameter yet.
8. Now complete the `ReviewPageComponent`:
    a. Add a `private route` parameter of type `ActivatedRoute` to the constructor.
    b. Inside the `ngOnInit()` method, assign the value of `this.route.snapshot.params['id']` to `this.bookId`.
9. Check the application still works and that the link now passes the id.

    Hint: If the page is blank and there are no error messages in the server console output, open Chrome Developer Tools and look for errors in the Console tab.
10. How about those tests?
    a. There are a number of unsatisfied dependencies.
11. Start with `BookListComponent`: fix up the missing item by adding the `RouterTestingModule` to the imports.
12. Move on to `ReviewPageComponent`.
    a. This can also be fixed by adding `RouterTestingModule` to the imports.
    b. Add an additional test using a mock `ActivatedRoute`.

Bonus Exercise (to be attempted if time permits)

13. Write Cypress E2E tests for the new Review page.

Chapter 14: Angular Forms

Exercise 14.1: Creating a Template-Driven Form

Time: 30 minutes

In this exercise, you will implement a template-driven form. The form will capture a review but will only log the results rather than save it to the web service.

1. Create `review.ts` inside the `models` folder. The `Review` class should have a `constructor` that declares two `public` properties, a `string content`, and a `number bookId`.

```
export class Review {
    constructor(
        public content: string,
        public bookId: number
    ) { }
}
```

2. Initial creation of the Review Form.
    a. Create a new `ReviewFormComponent` in the `reviews` folder. Since it is in the same folder, you can leave out the `--module` flag.

```
ng g component reviews/ReviewForm
```

    b. Add the selector to the `ReviewPageComponent` template after the line that says which book the reviews are for.
    c. Run the code and check that the dummy message appears where you expect it to.
3. Connect the Review Page and Review Form.
    a. Open the `ReviewFormComponent`.
    b. Add an `@Input()` property `bookId` of type `number`. (We looked at `@Input()` in Chapter 7 and Exercise 7.4. If you are uncertain, it might be worth reviewing the appropriate slide.)
    c. Add a `review` property of type `Review`.
    d. In the `ngOnInit`, initialize `review` to a `Review` object with `bookId` set to the `bookId` property and an empty `content`.

e. Open the `review-page.component.html` and pass the `bookId` in using a property binding.

f. Make a simple change to `review-form.component.html` so that the `bookId` is displayed. That will allow us to check that it works.

g. Make the `ReviewPageComponent` tests pass by adding a mock `ReviewFormComponent`. It must have an `@Input()` property to match the real form. Remember to add it to the `declarations` array.

```
@Component({
    selector: 'app-review-form',
    template: 'mock review form',
})
class ReviewFormMockComponent {
    @Input() bookId: number = -1;
}
```

4. Check that everything works and all the tests pass.

a. Your `ReviewFormComponent` might look a little like this:

```
export class ReviewFormComponent implements OnInit {

    @Input() bookId: number : -1;

    review: Review = new Review('',-1);

    constructor() { }

    ngOnInit() {
        this.review = new Review('', this.bookId);
    }
}
```

b. And the template:

```
<p>
  review-form works! {{bookId}}
</p>
```

c. While the template for `ReviewPageComponent` now looks like this:

```
<p>
    Review list for {{ bookId }}
</p>
<app-review-form [bookId]=bookId></app-review-form>
```

d. Yours may look different. The important things are the `@Input()` decoration and passing the data from the `ReviewPageComponent` to the `ReviewFormComponent`.

5. We will now create the form and add Angular Forms directives to it. Start by adding `FormsModule` to the list of imports for the `ReviewsModule`.
   a. You also need to add it to the test for `ReviewFormComponent`.
6. Now, replace the `ReviewFormComponent` template with an Angular form.
   a. The form should have a template reference variable `#revForm` assigned to `ngForm`.
   b. It should contain a `label` with the text `Review` and a text `input` with a two-way binding to `review.content`. Give the element the `name content` and add a template variable `#content` as the `ngModel`.
   c. Add a `submit` button with the text "Add Review".
   d. Add temporary test code after the form, using a property binding that displays `review.bookId` and `review.content` on the screen.
   e. The complete code for the template so far looks like this:

```
<form #revForm="ngForm">
    <label>Review:
        <input type="text" name="content" #content="ngModel"
            [(ngModel)]="review.content" />
    </label>
    <button type="submit">Add Review</button>
</form>
{{review.bookId + " " + review.content}}
```

7. In `review-form.component.spec.ts`, in the `TestBed` configuration, add an `imports` property for `FormsModule`:

```
TestBed.configureTestingModule({
    …,
    imports: [ FormsModule ]
});
```

8. Check your page again.
    a. Navigate to the reviews page, and type in the text box. You should see the output appearing on the page in your temporary test binding.
    b. You can see messages that you write with `console.log()` by opening Chrome Developer tools and clicking the Console tab.
    c. Take a look at your tests and make sure they all pass. They will because we are not testing the form yet!
9. Now, we need to do something with our review. Our service doesn't yet support reviews, so we will make a temporary change that logs the review to the console.
    a. Add a `submit()` method inside the `ReviewFormComponent` class. The method should simply `log() this.review` to the console.
    b. In the template, remove the temporary test binding code displaying the review properties on the page.
    c. Use a method binding to assign `submit()` to the `ngSubmit` directive.

```
<form #revForm="ngForm" (ngSubmit)="submit()">
```

10. Check it works. Open the developer tools to examine the console.

Bonus Exercise (to be attempted if time permits):

11. Style the form as best you can. You may wish to copy some styles from the Book Form, or you may want to make common styles in the `styles.css`.
12. Write Cypress E2E tests for the Review form.

Exercise 14.2: Testing a Template-Driven Form

Time: 20 minutes

In this exercise, you will add validation to your template-driven form and implement a test for that validation. Finally, you will tidy up functionality by clearing the form at the end.

1. Make the review content mandatory.
   a. Add the `required` attribute to the `input` element.
   b. Bind the `disabled` attribute of the `button` to code testing if the `form` is not `valid`.

```
<form #revForm="ngForm" (ngSubmit)="submit()">
    <label>Review:
        <input required type="text" name="content"
            #content="ngModel" [(ngModel)]="review.content" />
    </label>
    <button [disabled]="!revForm.form.valid"
        type="submit">Add Review</button>
</form>
```

2. Check your page. The button should be disabled until the user has entered something in the text field.
3. Add a test for your form to check the validation.

```
it('should validate content', async () => {
    await fixture.whenStable();

    const contentControl = fixture.debugElement
        .query(By.directive(NgForm))
        .references['revForm']
        .controls['content'];

    expect(contentControl.valid).toBeFalsy();
    expect(contentControl.hasError('required')).toBeTruthy();

    contentControl.setValue('a');

    fixture.detectChanges();
    await fixture.whenStable();

    expect(contentControl.valid).toBeTruthy();
});
```

4. Now we will add some user feedback.
    a. Add a new `div` either as the last child of the `form` element or as the first element after the form. We have added it after the form to make it easier to style the form and the message differently.
    b. Set the content of the `div` to the text `Review must have content`.
    c. Use an `*ngIf` directive to display the `div` only if `content` is not `valid`

```
<div *ngIf="!content.valid">Review must have content</div>
```

5. Improve the look and feel of your form by adding the following `styles` to the component level style sheet:

```
input.ng-valid {
    border-left: 5px solid #42A948;
}
input.ng-invalid {
    border-left: 5px solid #a94442;
}
```

6. Recheck the page and observe the behavior.
    a. The error message will be displayed if the control is invalid, regardless of whether it is pristine. It will disappear as soon as the control is valid.
    b. You should see a red bar to the left of the input when the content is invalid and green when it is valid.
7. Reset the form after the submit button has been pressed.
    a. Getting a reference to the form is difficult with a template-driven form. We could use `@ViewChild()`, but it is even easier to amend the `submit()` method to accept a parameter and pass in the template reference variable.
    b. The `ReviewFormComponent` method looks like this:

```
submit(form: NgForm) {
    console.log(this.review);
    form.resetForm();
}
```

    c. And the template looks like this:

```
<form #revForm="ngForm" (ngSubmit)="submit(revForm)">
```

Exercise 14.3: Creating a Model-Driven Form

Time: 30 minutes

In this exercise, you will revisit your `BookFormComponent` and replace it with a model-driven form.

1. Edit `books.module.ts`: replace `FormsModule` with `ReactiveFormsModule`. Make sure to fix up the file imports as well, removing the unused `FormsModule` import.
2. Open `book-form.component.html` and change the form to be model-driven.
    a. Replace the surrounding `div` with a `form`, or add a `form` if you didn't have any surrounding element before.
    b. The form opening tag should have an attribute `formGroup` bound to the value `bookForm` (we will create `bookForm` shortly).
    c. It should also have an event binding for `ngSubmit` to the existing `add()` method.
    d. Remove the `NgModel` bindings on the two input controls.
    e. Add `formControlName` directives to each input. Name them `title` and `author`, as appropriate.
    f. Change the type of the `button` to `submit`, remove the event binding, and set up the `disabled` property so the button is disabled when the form is invalid.
    g. Your HTML should look similar to this:

```
<form [formGroup]="bookForm" (ngSubmit)="add()">
  <label for="title">Title:</label>
  <input type="text" id="title" formControlName="title" />
  <label for="author">Author:</label>
  <input type="text" id="author" formControlName="author" />
  <button type="submit" [disabled]="!bookForm.valid">
    Add Book
  </button>
</form>
```

3. We must update the component to match. Open `book-form.component.ts`.
   a. Declare a `bookForm` property of type `FormGroup` and initialize it.

```
bookForm: FormGroup = new FormGroup({});
```

   b. Add a private formBuilder property of type FormBuilder to the constructor.
   c. Inside the `ngOnInit()` method, set `this.bookForm` equal to the return from a call to `this.formBuilder.group()`
   d. Add an object literal as the sole argument to `group()`. The object literal should have two properties, `title` and `author`. Each should be set to an array with two elements: an empty string and `Validators.required`

```
ngOnInit() {
    this.bookForm = this.formBuilder.group({
        title: ['', Validators.required],
        author: ['', Validators.required]
    });
}
```

   e. Change the `add()` method so that it emits a new book created with title and author from the form, along with an empty string (`cover`) and `-1` (`bookId`). Also have it reset the form. Notice how much easier it is to work with the model-driven form in code.

```
add() {
    this.createBook.emit(
        new Book(
            this.bookForm.get('title')?.value,
            this.bookForm.get('author')?.value,
            '',
            -1
        ));
    this.bookForm.reset();
}
```

   f. Delete the `book` property from the class.
4. Check that it works (you may need to use CTRL+SHIFT+R to force a reload of the page.) The Add Book button should only be enabled when there is content in both controls. Clicking the button should still add a book to the list.

5. Let's fix the existing tests. They no longer work because we switched from `FormsModule` to `ReactiveFormsModule` and removed the `book` property of the component class. Open `book-form.component.spec.ts`.

   a. The first change is to switch from `FormsModule` to `ReactiveFormsModule`. Fix the file imports as well.

   b. You should have one or two tests that check the event is correctly emitted when the button is pressed. We can replace them with a single test. Rather than update the book property directly, we can now interact with the form fields, which makes the test more effective. Remember that setting a control value programmatically does not mark the field as dirty, but fortunately, we are not checking the state of the fields.

```
it('should emit an event on click', () => {
  spyOn(component.createBook, 'emit');
  const expected = new Book('The Silmarillion',
                            'J R R Tolkien', '', -1);
  const form = fixture.debugElement.nativeElement
                      .querySelector('form');

  component.bookForm.get('title')?.setValue(expected.title);
  component.bookForm.get('author')?.setValue(expected.author);

  form.dispatchEvent(new Event('ngSubmit'));

  expect(component.createBook.emit)
        .toHaveBeenCalledWith(expected);
});
```

6. Now we should add a test for the form validation.

```
it('should validate content', () => {
    const titleCtrl = component.bookForm.get('title');
    const authorCtrl = component.bookForm.get('author');

    expect(component.bookForm.valid).toBeFalsy();
    expect(titleCtrl?.hasError('required')).toBeTruthy();
    expect(authorCtrl?.hasError('required')).toBeTruthy();

    titleCtrl?.setValue('The Inklings');
    authorCtrl?.setValue('Humphrey Carpenter');

    expect(component.bookForm.valid).toBeTruthy();
});
```

7. And let's add some visual feedback about validation.
    a. Inside the HTML template, add a `div` just after the form. Set its `class` attribute to your error message style, if you have one, and add the text `Title is required` as the content of the element.
    b. Add `*ngIf` to the `div`, checking `bookForm.get('title')?.hasError('required')`.
    c. Then create a second `div` that references the `author` rather than the `title`.

```
<div class="error"
    *ngIf="bookForm.get('title')?.hasError('required')">
    Title is required
</div>
<div class="error"
    *ngIf="bookForm.get('author')?.hasError('required')">
    Author is required
</div>
```

    d. Move your input styling from the review form to the root `styles.css`, which is available throughout the application.
    e. Do the same with any error message styles if you didn't already.
8. Test your page again.
    a. You should see validation messages from the beginning, regardless of whether the controls have been changed or not.
    b. You should see the controls styled to reflect whether their content is valid.

Bonus Exercise (to be attempted if time permits):

9. Improve the behavior of the validation messages by adding a second test to the `NgIf` so that the messages are only displayed if the matching control is not pristine.
10. Add a `minLength` validator for the book title. Add a suitable error message `div` in the template.
11. All these changes have probably broken the layout of your form, so fix it up.
12. Verify that your Cypress E2E tests all pass.

Exercise 14.4: Implementing Cross-Field Validation (Optional)

Time: 30 minutes

In this exercise, you will implement cross-field validation in the `BookFormComponent`. The validation will ensure that the `title` and `author` fields do not contain the same value.

1. We will start by creating the cross-field validator.
    a. Create a file in the shared folder. Name it `must-not-match-validator.ts`, or something similar. It is not a class file: we will not wrap the validator in a class this time.
    b. Also, create a corresponding spec file.
2. In the spec file, create a test fixture and a spec.
    a. Since this validator will ensure the title and author are different, we will need a `FormGroup` that contains a `title` and `author`. (We might typically use the `FormBuilder`, but we will just create them directly for now.)
    b. The spec will set the two controls to have the same value and check the `FormGroup` has the appropriate error.

```
import { titleAuthorMustNotMatch }
    from './must-not-match-validator';

describe('titleAuthorMustNotMatch', () => {
    let fg: FormGroup
    beforeEach(() => {
        fg = new FormGroup({
                title: new FormControl(''),
                author: new FormControl('')
            }, {
                validators: titleAuthorMustNotMatch
            });
    });
    it('should not allow control values to match', () => {
        fg.setValue({
            title: 'Dust',
            author: 'Dust'
        });
        expect(fg.valid).toBeFalsy();
        expect(fg.hasError('mustNotMatch')).toBeTruthy();
    });
});
```

3. Now implement enough code in the validator to pass the test.
    a. The signature of a validator is complex, so copy it from here:

```
export function titleAuthorMustNotMatch(
    control: AbstractControl): ValidationErrors | null { }
```

    b. The code should just return a suitable error object:

```
return { 'mustNotMatch': true };
```

4. Next, implement a test for two values that are not the same:

```
it('should allow control values to be different', () => {
    fg.setValue({
        title: 'Dust',
        author: 'Hugh Howey'
    });
    expect(fg.valid).toBeTruthy();
    expect(fg.hasError('mustNotMatch')).toBeFalsy();
});
```

5. And now the code to make both tests pass:

```
export function titleAuthorMustNotMatch(
        control: AbstractControl): ValidationErrors | null {
    return control.get('title')?.value ===
             control.get('author')?.value
         ? { 'mustNotMatch': true } : null;
};
```

6. We now have a working validator.
7. Next, we will add the cross-field validation to the form.
    a. First add it to the class in `book-form.component.ts`:

```
this.bookForm = this.formBuilder.group(
    {
        title: ['', [
            Validators.required,
            Validators.minLength(3)
        ]],
        author: ['', Validators.required]
    }, {
        validators: titleAuthorMustNotMatch
    });
```

b. Then add some visual feedback to the template:

```
<div class="error" *ngIf="!bookForm.pristine &&
bookForm.hasError('mustNotMatch')">
    Title and Author may not have the same value
</div>
```

8. Check out your form. It should now prevent the title and author having the same value.

Bonus Exercise (to be attempted if time permits):

9. Although the validation works, it leaves the uncomfortable situation where the form is invalid (error message, button disabled), but the controls show green bars because they are individually valid.

   a. Add an `NgClass` directive to the `<form>` to add an appropriate class when the cross-field validation has failed.

```
[ngClass]="{'form-invalid': !bookForm.pristine &&
bookForm.hasError('mustNotMatch')}"
```

   b. Define the class in the component stylesheet. This selector adds the style to any input controls inside an element having the `form-invalid` class.

```
.form-invalid input {
    border-left: 5px solid #a94442;
      }
```

This page intentionally left blank.

Chapter 15: Angular Deployment

Exercise 15.1: Lazy Loading a Feature Module (Optional)

Time: 15 minutes

In this exercise, you will convert the application to use lazy loading for the Reviews Module. We could add this for every feature module, but the Books Module is required when the application starts, so there would be a limited advantage in changing it.

1. We need to make changes to the `AppRoutingModule` so it lazy loads the `ReviewsModule` when appropriate.
    a. Add an entry to the Routes to specify a route for `/reviews`. (Remember that a route definition should not have the leading /.)
    b. This additional route should use `loadChildren` rather than specify a `component`.

```
{
    path: 'reviews',
    loadChildren: () => import('./reviews/reviews.module')
        .then(mod => mod.ReviewsModule)
}
```

2. Remove the import of `ReviewsModule` from the `AppModule`. Make sure to remove both the file import and the entry in the `imports` array.
3. Amend the route definition in the `ReviewsRoutingModule`.
    a. The `AppRoutingModule` already defines the `/reviews` part of the route, so we can remove that. (Again, remember that a route definition should not have the leading /.)

```
const routes: Routes = [{
    path: ':id',
    component: ReviewPageComponent
}];
```

4. Run the application and confirm that you can still navigate from the book list to a review.

Exercise 15.2: Building and Deploying the Application

Time: 30 minutes

In this exercise, you will build and deploy the application.

1. We will eventually deploy the application as the `BookStore` web application.
   a. The application will be served from http://localhost:8080/**BookStore**
   b. The service will be served from http://localhost:8080/BookService/jaxrs
2. Enable the live server proxy.
   a. Create a file `proxy.conf.json` and set the redirects so the Angular application can talk to `/BookService/jaxrs` and the service can run on `localhost:8080`:

```
{
    "/": {
        "target": "http://localhost:8080",
        "secure": false,
        "logLevel": "debug"
    }
}
```

   b. Edit the `angular.json` file and add an `options` element with a `proxyConfig` setting pointing to the new proxy config file:

```
{
   …
   "projects": {
      "BookStore": {
         …
            "architect": {
               …
               "serve": {
                  "builder": "…",
                     "options": {
                        "proxyConfig": "src/proxy.conf.json"
                     },
                     …
```

3. Edit the `BookService` to use a different URL:

```
public readonly url: string = '/BookService/jaxrs/books';
```

4. When our app is deployed to a server, it needs help managing the URLs for images. To get that help, we will inject a `LocationStrategy` into the `NoImagePipe`. Then, we will use the `LocationStrategy` to return a URL that works on the server.

    a. Add a constructor to the `NoImagePipe` class and modify its `transform()` method as follows:

```
export class NoImagePipe implements PipeTransform {

  constructor(private locationStrategy: LocationStrategy) {
}

  transform(path: string): string {
    return path ? path
                : this.locationStrategy.getBaseHref() +
                  'assets/images/NoImage.svg';
    }
}
```

5. At this point, verify that the application runs.
6. Shut down the application and the unit tests.
7. Next, build the application with the development configuration:

```
ng build –configuration=development
```

8. Examine the `dist/BookStore` subdirectory.

    a. Open one of the files whose name starts with `main`. Can you find any of your components in it? Compare the transpiled code with your TypeScript code.

    b. Make a note of the file names and sizes.

9. Build the application for production. (*Note:* don't run this command in a `bash` prompt because it causes problems with the `base-href` setting.)

```
ng build --configuration=production --base-href=/BookStore/
```

10. Examine the `dist/BookStore` sub-directory again.

    a. Open one of the files whose names start with `main` again. Can you find any of your components in it?

    b. Compare the file names and sizes with your previous list.

11. Now you can deploy the Angular application to the Tomcat server. Copy the entire `dist/BookStore` folder into the Tomcat folder `C:\Apps\tomcat\webapps`.
    a. Tomcat will now serve the Angular app, so you don't need to run `npm start` or `ng serve`.
12. The application is available at `localhost:8080/`**`BookStore`**. Verify that it works as before.

Bonus Exercise (to be attempted if time permits)

13. It is tedious and unreliable to have to update the service URL in the `BookService`. Let's add an item to the `environment`.
    a. Edit the `environment.ts` file.
    b. Create an additional property: `serviceUrl` with a value `/BookService/jaxrs/books`.

```
export const environment = {
    production: false,
    serviceUrl: '/BookService/jaxrs/books'
};
```

    c. Add the same property with the same value to `environment.development.ts`.
    d. We could have different values in development and production, but we don't need that.
14. Amend the `BookService` so that it uses the service URL from the environment instead of hardcoding the URL.

```
import { environment } from '../environments/environment';
…
public readonly url: string = environment.serviceUrl;
```

15. Check that the tests and the application still work.

    Try re-deploying the application to the Tomcat server.

Appendix A: Angular Directives

Exercise A.1: Creating an Attribute Directive

Time: 30 minutes

This exercise is not part of the course, but it allows you to practice the material covered in Appendix A.

In this exercise, you will create an attribute directive that enables dragging over and dropping onto an HTML element.

Your directive will allow images to be drop-targets but will not itself implement the `drop` event. The directive's job is to make drag and drop possible, not to decide what should happen when something is dropped. At a minimum, you will need to do three things inside the directive: prevent the default behavior for the `dragover` and `drop` events, create an event to let client code know that something has been dropped, and passing through the event object.

1. Using the CLI, generate a new directive:

   ```
   ng generate directive shared/drag-drop --module shared
   ```

2. Add the directive to the `exports` array of the `SharedModule`.
3. Open the directive file.
   a. Add a `constructor` that injects a `private` property `el` of type `elementRef`
   b. Define an `@Output() dropped` of type `EventEmitter<DragEvent>` and set it equal to a `new EventEmitter()`. Make sure that `EventEmitter` is imported from `@angular/core`, rather than anywhere else.
   c. Next, add a method `onDragOver()` to the class. The method should accept a single argument `event` of type `DragEvent`. Inside the body of the method, call `preventDefault()` on the `event` object.
   d. Decorate the `onDragOver()` method with a `@HostListener()` for the `dragover` event. The second argument to `@HostListener()` should be an array with a single string element `$event`.
   e. Create an `onDrop()` host listener. This should be the same as the `onDragOver()` method, but amended to refer to the `drop` event.

f. In addition, the last line of the `onDrop()` method should emit your `dropped` event, passing in the `event` object as the sole argument.

g. The code of your directive should look like this:

```
import { Directive, ElementRef, Output, EventEmitter,
    HostListener } from '@angular/core';

@Directive({
    selector: '[appDragDrop]'
})
export class DragDropDirective {

    @Output()
    dropped: EventEmitter<DragEvent> = new EventEmitter();

    constructor(private el: ElementRef) { }

    @HostListener('dragover', ['$event'])
    onDragOver(event: DragEvent) {
        event.preventDefault();
    }

    @HostListener('drop', ['$event'])
    onDrop(event: DragEvent) {
        event.preventDefault();
        this.dropped.emit(event);
    }
}
```

4. Open the `BookFormComponent`.

a. Add a property `cover` and initialize it to an empty string.

b. In the `add()` method, change the `book` constructor to use `this.cover` rather than an empty string.

c. Add a new method `onDrop()` that accepts a single parameter `event` of type `DragEvent`.

d. Put the following code in the method:

```
onDrop(event: DragEvent) {
    const files = event.dataTransfer.files;
    console.log(files);
}
```

e. Note that this code will work in modern browsers, but it is not necessarily safe in older ones.

5. Open the `BookFormComponent` template.
   a. Before the button, add an `img` element.
   b. The `img` element should have the `appDragDrop` directive.
   c. It should also have a method binding to assign `onDrop($event)` as the handler for the `dropped` event.
   d. And the `src` attribute should be set to an interpolation binding of the cover property using the noImage pipe.

```
<img appDragDrop (dropped)="onDrop($event)"
    src="{{ cover | noImage }}" />
```

6. Open the `BookFormComponent` styles (`book-form.component.css`) and add a rule that restricts `img` elements to a `max-width` and `max-height` of 100px.
7. Check that your application works. If you drag a file over the drop-target, the details will be logged to the console.
8. Fix any tests that are not working.
   a. You should add a drag-drop directive to the `BookFormComponent` test. The directive does so little that it could be the real one, but it would be more sensible to add a mock. Also add it to the `declarations` array.

```
@Directive({
    selector: '[appDragDrop]'
})
export class MockDragDropDirective { }
```

   b. The component also now uses `NoImagePipe`. We have a mock version for use in the `BookListComponent` tests. Let's extract that into a separate file to make it reusable.
   c. Create a new folder under `src/app`, call it `mocks`.
   d. Create a new file in mocks called `mock-no-image-pipe.ts`. We could create this using the Angular CLI, but it is just as easy to create it manually since we don't want to add it to any Module.

e. Take the mock code from the tests for `BookListComponent` and paste it into the new file. Ensure the class is exported.

```
import { Pipe, PipeTransform } from "@angular/core";

@Pipe({
    name: 'noImage'
})
export class MockNoImagePipe implements PipeTransform {
    transform(value: string): string {
        return '';
    }
}
```

f. Add `MockNoImagePipe` to the `BookFormComponent` tests.

g. Update `BookListComponent` tests to use the new shared mock as well.

9. Create a test for the new directive.

a. Create a dummy component that uses the directive:

```
@Component({
    selector: 'dummy',
    template: `<div appDragDrop (dropped)="onDrop($event)">
        aaa</div>`
})
class TestDragDropComponent {
    @ViewChild(DragDropDirective, { static: false })
    directive: DragDropDirective;

    onDrop(event: DragEvent) {
        // do nothing
    }
}
```

b. Write a test to see whether the component's `onDrop()` method is invoked when something is dropped on the div:

```
it('should emit event on drop', () => {
    // Allows normal functionality to continue
    spyOn(component.directive.dropped, 'emit')
        .and.callThrough();
    spyOn(component, 'onDrop');
    const divEl = fixture.debugElement.query(By.css('div'));
    divEl.triggerEventHandler('drop',
        new DragEvent('drop'));
    expect(component.directive.dropped.emit)
        .toHaveBeenCalled();
    expect(component.onDrop).toHaveBeenCalled();
});
```

Bonus Exercise (to be attempted if time permits):

10. Improve the behavior of your directive by giving visual feedback when a cover image is dragged over the image element.

    a. Use an `@HostBinding()` to link a property of the class to a property of the host element. Adding a border is popular but prefer outline to border since it doesn't change the physical dimensions of the element. You could try opacity or the brightness filter.

    ```
    @HostBinding('style.opacity')
    opacity: string;
    ```

    b. Apply the effect when the user moves the mouse over the host element.

    c. Clear the effect when the user drops the image or if the user moves off the element. You will need to add a `HostListener` for the `dragleave` event to respond to the user moving off the drop target.

11. If you want drop to work correctly, add the `ImageLoaderService`.

    a. Copy the service source code from the `AppendixA` directory. Put it in the `src/app` folder. Note that it isn't possible to write a meaningful test for the service as it stands since it uses the browser File API: for security reasons, the file cannot be set programmatically.

    b. Inject the service into the `BookFormComponent` constructor.

c. Replace the `onDrop()` method with this code:

```
onDrop(event: DragEvent) {
    const files = event.dataTransfer.files;
    if (files && files.length > 0) {
        // only interested in one file
        this.imageLoader.onDropImageFile(files[0])
            .then((result: string) => this.cover = result);
    }
}
```

12. Fix the form `reset()` issue by adding a line in the `add()` method resetting `this.cover` to an empty string literal.

Appendix B: Observables

Exercise B.1: Making RESTful Calls Using Observables

Time: 40 minutes

This exercise is not part of the course, but it allows you to practice the material covered in Appendix B.

In this exercise, you will use Observables to make the system more reactive.

1. Convert `BookListComponent` to use the `AsyncPipe`.
    a. Change the declaration of `books` in `BookListComponent` from `Book[]` to `Observable<Book[]>`.
    b. In the `BookListComponent` template, change the `NgFor` to use the `waitForAsync` pipe. The `trackBy` must occur *after* the `waitForAsync`.
    c. In the `BookPageComponent`, change books in the same way (to be an `Observable<Book[]>`) and change `getBooks` so that `books` is assigned the result of `bookService.getBooks()`. There should no longer be a `subscribe`.
    d. If you have an additional `div` in the `BookListComponent` that is displayed when there are no books, comment it out for now.
2. Check that everything works.
3. What happened to the tests? It was inconceivable that such a major change could be accomplished without having a significant impact on the tests.
    a. The `BookListComponent` now has an input property that is an Observable, so the mock version in the `BookPageComponent` tests needs to reflect that.
    b. An Observable does not have a simple length property, so some of the tests will now need to subscribe to the observable and test the length of the data instead. The easiest way to do this is to make them `fakeAsync()` tests.

c. For example:

```
it('should pass books to the child component',
        fakeAsync(() => {
    const bookList = fixture.debugElement.query(By
        .css('app-book-list')).componentInstance;
    let books: Book[];
    bookList.books
        .subscribe(data => books = data)
    tick();
    expect(books.length).toBe(2);
}));
```

d. And:

```
it('should retrieve books from the service',
        fakeAsync(() => {
    let books: Book[];
    component.books
        .subscribe(data => books = data)
    tick();
    expect(books.length).toBe(2);
    expect(books[0].title).toBe('The Hobbit');
    expect(books[1].title).toBe('A Wizard of Earthsea');
}));
```

e. In the `BookListComponent`, wherever the tests assign a list of books to `component.books`, they now need to assign an `Observable`. Create one from the list of books using `of()`.

f. For now, comment out the test that checks for the special `div` for an empty book list.

4. You will now add a feature to allow users to search by book title.

5. Add a method `search()` to the `BookPageComponent`.

   a. It should accept a single argument `term`, of type `string`.

   b. It should log the `term` to the console.

6. Add the user interface for search to the `BookPageComponent` template.

   a. Create a `div` between the book form and the book list.

   b. Add an `input` inside the `div` with the attribute `#term` and use a method binding on the `keyup` event to pass `term.value` to `search()`.

c. Add a `label` with suitable text.

```
<div>
    <label>Search by title:
        <input #term (keyup)="search(term.value)" />
    </label>
</div>
```

d. We should really create a new component to do this, but for simplicity, we will do it in the `BookPageComponent`.

7. At this point your code should work, but only log the search term to the console.
8. Create a new method `getBooksByTitle()` in the `BookService`.
    a. The method should accept a single argument of type `string`. And return `Observable<Book[]>`.
    b. You need to test that your method correctly passes the parameters to the `HttpClient`. Note that special characters (like space) in the parameter will be escaped (space becomes %20). Your test might look like this:

```
it('should search for books', inject([BookService],
        fakeAsync((service: BookService) => {
    let books: Book[];
    service.getBooksByTitle('the l')
        .subscribe(data => books = data);
        const req = httpTestingController
            .expectOne(service.url + '?title=the%20l');

    // Assert that the request is a GET.
    expect(req.request.method).toEqual('GET');

    // Respond with mock data, causing Observable to resolve.
    req.flush(testBooks);

    tick();
    expect(books).toBeTruthy();
    expect(books[0].title).toBe('The Lord of the Rings');
})));
```

    c. Working TDD, create the code for your method.
    d. The URL takes an additional query parameter:

```
…/books?title=<query string>
```

e. Use an `HttpParams` object rather than building this up through string concatenation, since this will guarantee that the string is properly escaped:

```
return this.http.get<Book[]>(this.url, {
    params: new HttpParams().set('title', title)
}).pipe(catchError(this.handleError));
```

f. Technically, you could bypass the parameter when there is no search term, but the service works equally in either case, so that isn't necessary.

9. Now, we will connect all this in the `BookPageComponent`.

a. Remove `getBooks()` and the code in `ngOnInit()` that calls it. You may remove the `ngOnInit` and `implements` clause completely, if you want. If you do, remove the import as well.

b. Before the definition of books, create a new property `searchStream` of type `BehaviorSubject<string>`. Set it equal to a new `BehaviorSubject<string>` of an empty string.

c. Change the declaration of books so that it is now:

```
books: Observable<Book[]> = this.searchStream
    .pipe(switchMap((term: string) =>
        this.bookService.getBooksByTitle(term)));
```

d. You will need to import `switchMap` from `rxjs/operators`.

e. Change your search method so that it passes `term` to the `next()` method of `searchStream`. You can decide whether or not to remove the logging.

f. Change `addBook()` to use `search()` instead of `getBooks()`. Pass an empty string.

g. You will need to fix up tests for `BookPageComponent`. The fake `BookService` implements `getBooks()` but we are now using `getBooksByTitle()`. Otherwise, the definition can remain the same.

10. Check your code to see that it works and all your tests pass.

a. Note that initializing the `BehaviorSubject` with an empty string has caused all books to be retrieved.

11. Let's reduce the number of network calls.
    a. Add `debounceTime(500)` to the `searchStream pipe` function call ahead of `switchMap`.

```
books: Observable<Book[]> = this.searchStream
    .pipe(
        debounceTime(500),
        switchMap((term: string) =>
            this.bookService.getBooksByTitle(term))
    );
```

    b. And the tests? They fail again. In this case, because of the delay introduced by `debounceTime()`. Change the calls to `tick()` to `tick(500)`.

12. Check your code again. It should work as before, but it should issue fewer network calls by waiting for the user to pause.

Bonus Exercise (to be attempted if time permits):

13. Now let's put back the handling for an empty list of books.
    a. There is no easy way to use the `AsyncPipe` twice on the same `Observable`, so the simplest way to do this is to "tap" the `Observable` to look at the value without affecting it.
    b. We will add a new `@Input()` property to the `BookListComponent` to accept a `boolean` indicating whether there are any books or not.

```
@Input() nobooks: boolean;
```

    c. Add a property to the `BookPageComponent`.

```
nobooks = true;
```

    d. Populate it using a tap.

```
books: Observable<Book[]> = this.searchStream
    .pipe(
        debounceTime(500),
        switchMap((term: string) =>
            this.bookService.getBooksByTitle(term)),
        tap(data => this.nobooks = (data.length == 0))
    );
```

e. And a binding in the `BookPageComponent` template.

```
<app-book-list [books]="books"
    [nobooks]="nobooks"></app-book-list>
```

f. In the `BookListComponent` template, change the condition for the `div`.

```
<div id="nobooks" *ngIf="nobooks">
    There are no books available
</div>
```

14. If you type something in the search control and there are no books starting with that text, you should see the message.
15. It has made a bit of a mess of the tests because there is an additional input binding required. Update the mock `BookListComponent` from the `BookPageComponent` tests so it supports the additional `@Input()`.
16. In the tests for `BookListComponent`, uncomment the test for the no data `div`. The test will need to change to use `nobooks` instead of `books`.
17. There should be a test of `BookPageComponent` checking that it sets `nobooks` to `true` when there are no books and to `false` when there are books in the list.
    a. This is complicated by the fact that there is already a fake for the `getBooksByTitle()` method. The simplest way is to create a completely new describe.
    b. Start by testing for the negative case. Add a check for `nobooks` being `false` to the test that checks the list is passed to the child component. You will need a `fixture.detectChanges()` to update bindings.

```
it('should pass books to the child component',
        fakeAsync(() => {
    const bookList = fixture.debugElement.query(By
        .css('app-book-list')).componentInstance;
    let books: Book[];
    bookList.books
        .subscribe(data => books = data)
    tick(500);
    expect(books.length).toBe(2);
    fixture.detectChanges();
    expect(bookList.nobooks).toBe(false);
}));
```

c. Now add the new describe for the situation where there is no data.

```
describe('BookPageComponent', () => {
    let component: BookPageComponent;
    let fixture: ComponentFixture<BookPageComponent>;

    beforeEach(waitForAsync(() => {
        // A test list of books
        const testBooks: Book[] = [];
        // Create a fake BookService object
        const bookService = jasmine.createSpyObj(
            'BookService', ['getBooksByTitle']);
        bookService.getBooksByTitle
            .and.returnValue(of(testBooks));

        TestBed.configureTestingModule({
            declarations: [
                BookPageComponent,
                MockBookListComponent,
                MockBookFormComponent
            ],
            providers: [
                { provide: BookService,
                    useValue: bookService }
            ]
        })
            .compileComponents();
    }));

    beforeEach(() => {
        fixture = TestBed
            .createComponent(BookPageComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
    });

    it('should pass nobooks to child if there are no books',
            fakeAsync(() => {
        let books: Book[];
        const bookList = fixture.debugElement.query(By
            .css('app-book-list')).componentInstance;
        component.books
            .subscribe(data => books = data)
        tick(500);
        expect(books.length).toBe(0);
        fixture.detectChanges();
        expect(bookList.nobooks).toBe(true);
    }));
});
```

18. There is currently no error handling.
    a. It used to be there, but because we are not doing integration testing, we didn't notice that it was effectively disabled.
    b. We can add error handling to the `pipe()` in the definition of the `books Observable` in `BookPageComponent`, but we also need somewhere to reset the message. We will use the existing `tap()` to do that.

```
books: Observable<Book[]> = this.searchStream
    .pipe(
        debounceTime(500),
        switchMap((term: string) =>
            this.bookService.getBooksByTitle(term)),
        tap(data => {
            this.errorMessage = '';
            this.nobooks = (data.length == 0)
        }),
        catchError(error => {
            this.errorMessage = error;
            return of([]);
        })
    );
```

19. Let's see what happens when an error appears and then goes away.
    a. Change the `getBooksByTitle()` method of the service so that it sometimes throws an error.

```
getBooksByTitle(title: string): Observable<Book[]> {
    let url = this.url;
    if (title == 'c') {
        url = 'z';
    }
    return this.http.get<Book[]>(url, {
        params: new HttpParams().set('title', title)
    }).pipe(catchError(this.handleError));
}
```

    b. Now whenever the title is 'c', the service will throw an error.
    c. Try it out. You should see the error message when you type 'c'. What happens when you delete the 'c'?
    d. It stops!

e. This is a fundamental feature of `Observable`: as soon as it hits an error, the stream stops. But we can make it retry.

```
books: Observable<Book[]> = this.searchStream
    .pipe(
        debounceTime(500),
        switchMap((term: string) =>
            this.bookService.getBooksByTitle(term)),
        tap(data => {
            this.errorMessage = '';
            this.nobooks = (data.length == 0)
        }),
        retryWhen(errors => {
            return errors
                .pipe(
                    tap(data => this.errorMessage = data),
                    delayWhen(() => timer(2000)),
                    tap(() => console.log('retrying...'))
                );
        })
    );
```

f. Now you should see that the error resets itself. There are other retry options available.

g. Once you are happy, fix the method in the service so that it no longer throws an error.

20. Tidy up the presentation.

Exercise B.2: Functional Reactive Forms and Observables

Time: 20 minutes

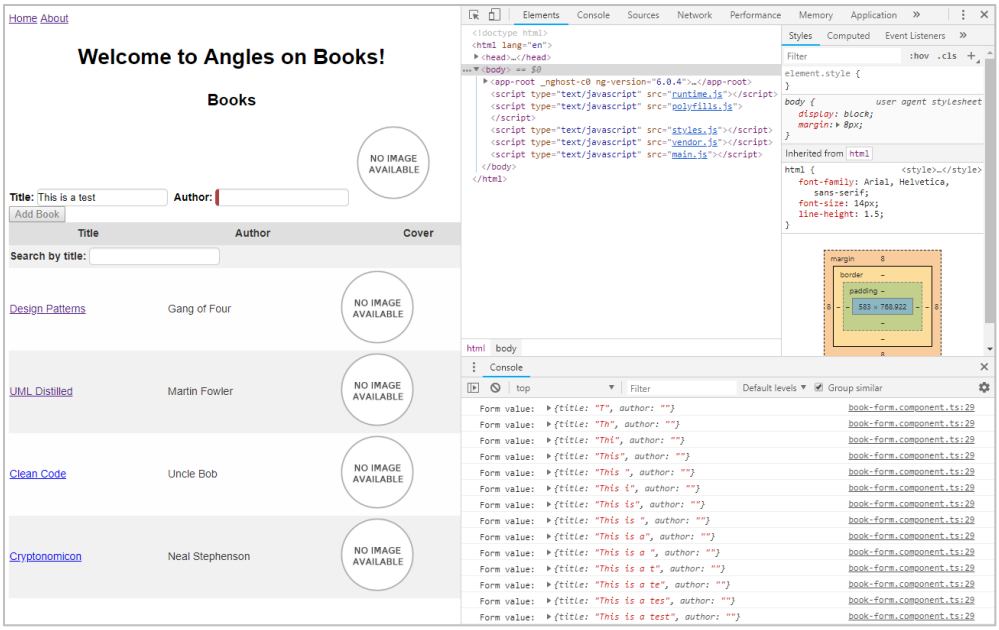
This exercise is not part of the course, but it allows you to practice the material covered in Appendix B.

In this exercise, you will modify your existing book form to use streams.

1. You will now change the book form so that it responds to key presses. You could do validation, but for test purposes we will just convert the string to title case.
2. Open `BookFormComponent`.
    a. Add a property `sub` (of type `Subscription`) to the class.
    b. At the end of the `ngOnInit()` method, set `sub` equal to a subscription to the form's value changes and log the value to the console.

```
this.sub = this.bookForm.valueChanges.subscribe(value => {
    console.log("Form value: ", value);
});
```

    c. Add `OnDestroy` to the list of interfaces implemented and in the `ngOnDestroy()` method, unsubscribe from the subscription.
3. Check that everything runs OK. As you type in the book form, you should see console events.



4. Now, you will convert to title case. Return to the `BookFormComponent`.
    a. Define a new method `toTitleCase()`.

```
toTitleCase(s: string) {
    if ((s === null) || (s === '')) {
        return '';
    }
    return s.replace(/\w\S*/g, (t: string) => {
        return t.charAt(0).toUpperCase()
            + t.substr(1).toLowerCase();
    });
}
```

    b. Now, change the arrow function in the subscription. Above the logging code, make a call to `this.toTitleCase`, passing in `value.title` and putting the result back in `value.title`.
5. Check that your code works, and you see items logged in title case. Nothing changes in the form yet.
6. Now, go back to `BookFormComponent`.
    a. At the end of the subscribe arrow function, make a call to `patchValue` on `bookForm`.

```
this.bookForm.patchValue({ title: value.title },
    { emitEvent: false });
```

7. Now, you should see the value in the form change as well as being logged. Once you are happy, you can remove the logging.