

DEVELOPING CLIENT-SIDE DYNAMIC WEB APPLICATIONS

EXERCISE MANUAL



This page intentionally left blank.

Table of Contents

GENERAL INSTRUCTIONS	1
CHAPTER 1: INTRODUCTION TO VISUAL STUDIO CODE	3
EXERCISE 1.1: INTRODUCTION TO VISUAL STUDIO CODE	3
GETTING TO KNOW VISUAL STUDIO CODE (VSC)	3
GENERATE YOUR FIRST WEB PAGE WITH VSC	4
EXECUTING WEB PAGE	5
CHAPTER 2: STANDARDIZING PRESENTATION WITH HTML AND CSS	7
EXERCISE 2.1: APPLYING CSS STYLING	7
CHAPTER 3: ADVANCED HTML AND CSS	9
EXERCISE 3.1: APPLYING ADVANCED CSS STYLING EFFECTS	9
OPTIONAL EXERCISE 3.2: STATIC WEB PAGES	10
EXERCISE 3.3: CREATING AN HTML5 PAGE USING SEMANTIC TAGS	11
EXERCISE 3.4: CREATING AN HTML FORM WITH VALIDATION	13
CHAPTER 4: CLIENT-SIDE JAVASCRIPT PROGRAMMING	15
EXERCISE 4.1: JUMPING JAVASCRIPT	15
EXERCISE 4.2: JAVASCRIPT ARRAYS AND OBJECTS	16
EXERCISE 4.3: MANIPULATING THE DOM	17
EXERCISE 4.4: WORKING WITH BUILT-IN CLASSES	18
EXERCISE 4.5: RESPONDING TO EVENTS	19
EXERCISE 4.6: FORM VALIDATION WITH JAVASCRIPT (OPTIONAL)	20
VALIDATION IN THE CLICK EVENT	20
VALIDATION IN THE SUBMIT EVENT	20
CHAPTER 5: WORKING WITH JQUERY	23
EXERCISE 5.1: FIRST STEPS WITH JQUERY	23
EXERCISE 5.2: PUTTING IT ALL TOGETHER	24
EXERCISE 5.3: AJAX WITH JQUERY	25
START THE SIMPLESERVER	25
COMPLETE THE AJAX CLIENT	25
CHAPTER 6: INTRODUCTION TO ANGULAR	27
EXERCISE 6.1: GETTING STARTED WITH ANGULAR	27
EXERCISE 6.2: WRITE YOUR FIRST TEST SPECS	30
EXERCISE 6.3: UNIT TESTING ANGULAR	31
CHAPTER 7: ANGULAR COMPONENTS	33
EXERCISE 7.1: CREATING A COMPONENT	33

EXERCISE 7.2: UNIT TESTING A COMPONENT	37
EXERCISE 7.3: USING BUILT-IN DIRECTIVES	39
EXERCISE 7.4: REFACTORING COMPONENTS	41
CHAPTER 8: ANGULAR MODULES AND BINDING.....	43
EXERCISE 8.1: CREATING A MODULE	43
EXERCISE 8.2: TWO-WAY AND EVENT BINDING.....	44
CHAPTER 9: PIPES	51
EXERCISE 9.1: CREATING A CUSTOM PIPE.....	51
CHAPTER 10: ANGULAR SERVICES.....	55
EXERCISE 10.1: CREATING AND INJECTING A SERVICE.....	55
EXERCISE 10.2: RETRIEVING AND ADDING DATA WITH REST	61
DEPLOY THE BOOKSERVICE ON TOMCAT	61
BUILD THE ANGULAR APPLICATION.....	62
EXERCISE 10.3: HANDLING ERRORS IN A RESTFUL SERVICE (OPTIONAL)	65
CHAPTER 11: BUILDING AN APPLICATION.....	69
EXERCISE 11.1: BUILDING AN ANGULAR APPLICATION	69
BUILD THE APPLICATION	69
THE CAR CLASS.....	70
MAKE THE CARLISTCOMPONENT.....	71
CREATE THE CARS SERVICE.....	71
CONVERT CARSERVICE TO USE HTTP	72
ADD THE BUTTONS.....	73
CHAPTER 12: ANGULAR ROUTING	75
EXERCISE 12.1: ROUTING WITH THE ANGULAR ROUTER.....	75
EXERCISE 12.2: PASSING AND RECEIVING ROUTE PARAMETERS.....	78
CHAPTER 13: ANGULAR FORMS	81
EXERCISE 13.1: CREATING A TEMPLATE-DRIVEN FORM.....	81
EXERCISE 13.2: TESTING A TEMPLATE-DRIVEN FORM	85
EXERCISE 13.3: CREATING A MODEL-DRIVEN FORM	89
EXERCISE 13.4: IMPLEMENTING CROSS-FIELD VALIDATION (OPTIONAL).....	93
CHAPTER 14: ANGULAR END-TO-END (E2) TESTING APPLICATIONS.....	97
EXERCISE 14.1: DESIGNING E2E TESTING.....	97
EXERCISE 14.2: WRITING A SIMPLE PROTRACTOR TEST.....	98
EXERCISE 14.3: ENTERING DATA IN E2E TESTS	100
EXERCISE 14.4: WRITING AN E2E TEST FOR MANAGECARS (OPTIONAL)	103

CHAPTER 15: ANGULAR DEPLOYMENT	105
EXERCISE 15.1: LAZY LOADING A FEATURE MODULE (OPTIONAL)	105
EXERCISE 15.2: BUILDING AND DEPLOYING THE APPLICATION	106
APPENDIX A: ANGULAR DIRECTIVES	111
EXERCISE A.1: CREATING AN ATTRIBUTE DIRECTIVE	111
APPENDIX B: OBSERVABLES	117
EXERCISE B.1: MAKING RESTFUL CALLS USING OBSERVABLES	117
EXERCISE B.2: FUNCTIONAL REACTIVE FORMS AND OBSERVABLES	126

This page intentionally left blank.

General Instructions

Exercises are done by an individual student or in pairs. Workshops are done by assigned workgroups on flip charts or whiteboards provided.

This page intentionally left blank.

Chapter 1: Introduction to Visual Studio Code

Exercise 1.1: Introduction to Visual Studio Code

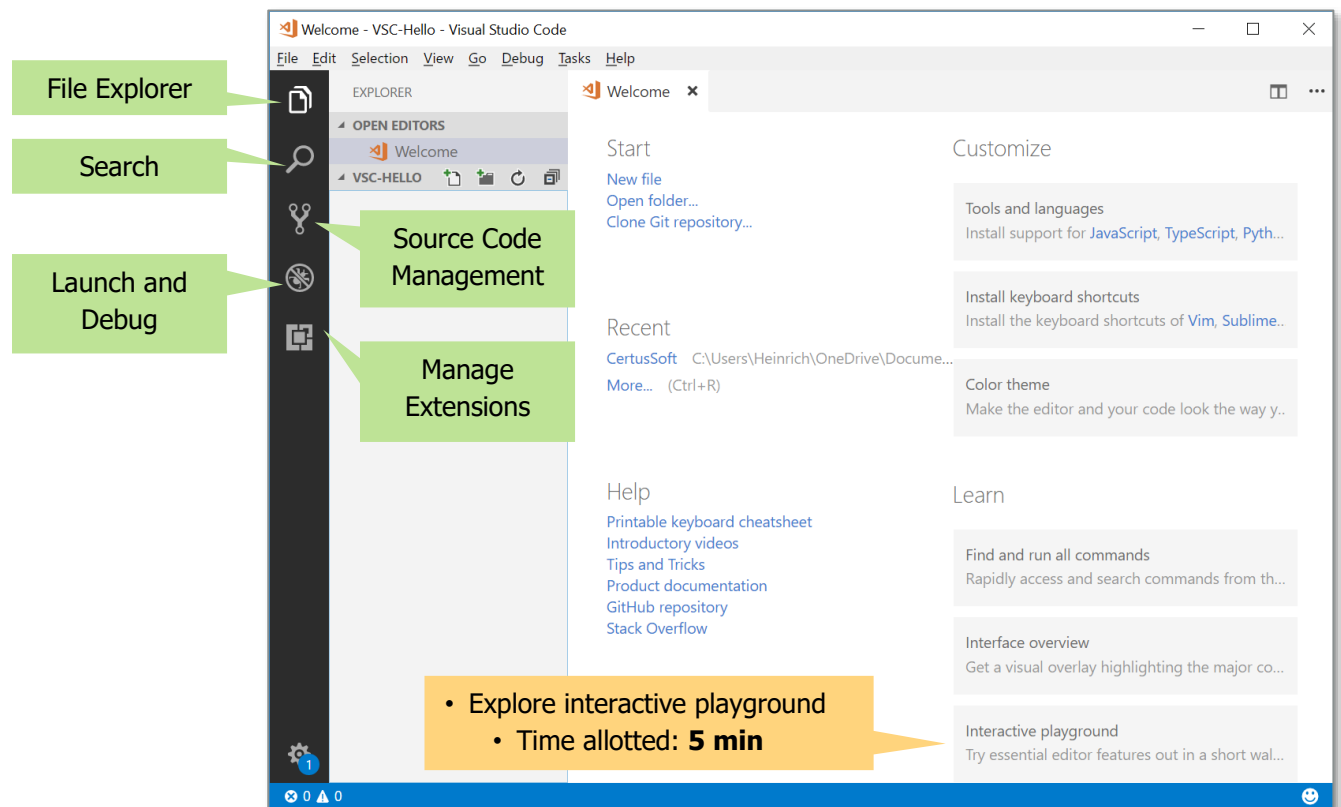
Time: 40 minutes

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)

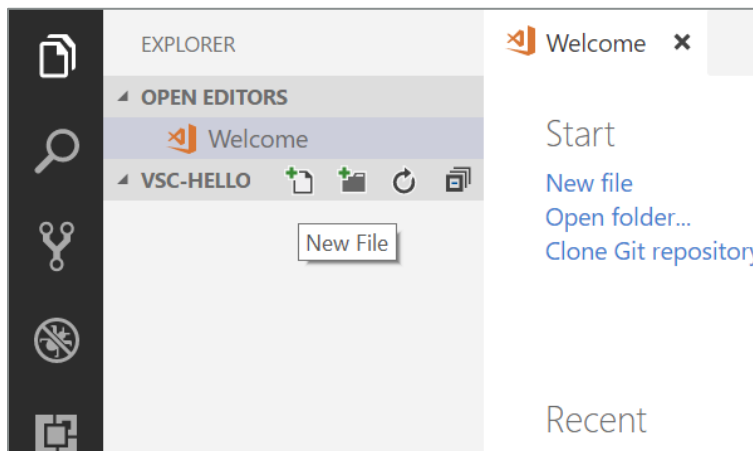
1. Start Visual Studio Code. The Welcome page should be displayed. Your instructor will point out several features of the VSC environment.



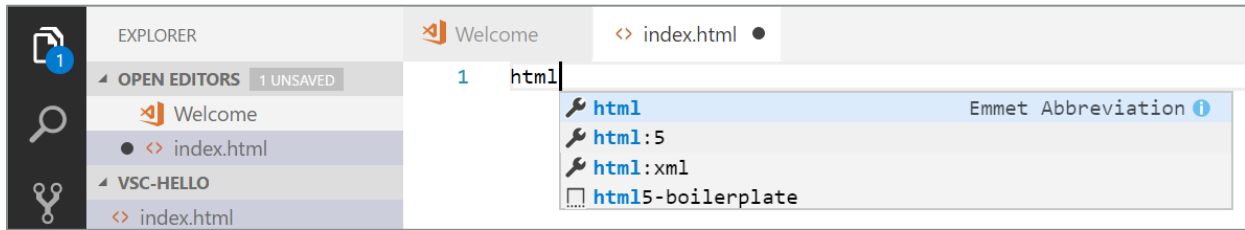
2. Click the **Manage Extensions** icon.
 - a. In the search field, type: `html boilerplate`.
 - b. Select **HTML Boilerplate** from the list.
 - c. Click **Install**.
3. Do the same with:
 - a. Open in browser
 - A quick way of opening the current file in your default browser.
 - b. Debugger for Chrome
 - Allows you to debug your code in Chrome.
 - Version 4.11 or higher.
 - c. Live Server
 - Launches a web page in a development server.
 - d. CSS Formatter
 - Choose the one by Martin Aeschlimann.
 - Allows the same formatting of CSS as VS Code already provides for JavaScript and HTML.
 - e. Explore what's on offer

Generate Your First Web Page with VSC

4. Choose work folder: `WebAppsClient\Ch01\VSC-Hello`.
 - a. Hover over the `VSC-Hello` bar and click **New File** icon.
 - Name it: `index.html`.

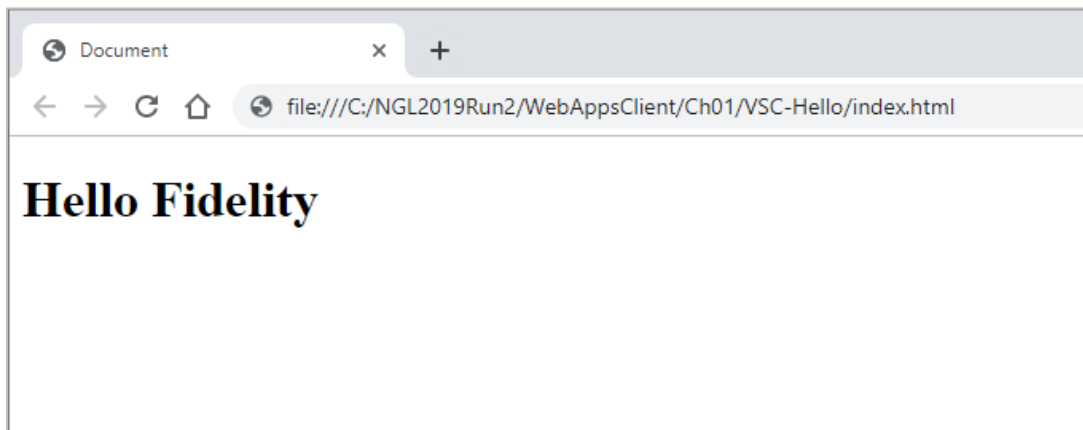


5. Switch to the `index.html` tab, type `html`, and click **html:5** ->
 - a. Boilerplate code is implemented to create a basic page structure.



Executing Web Page

6. First, add a heading to the web page:
 - a. Add `<h1>Hello Fidelity</h1>` between the `<body>` tags.
 - b. Save file.
 - c. Right-click the file in the VS Code Explorer and choose **Open in Default Browser**.
 - You should see something like the following:



This page intentionally left blank.

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 flow of text 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.

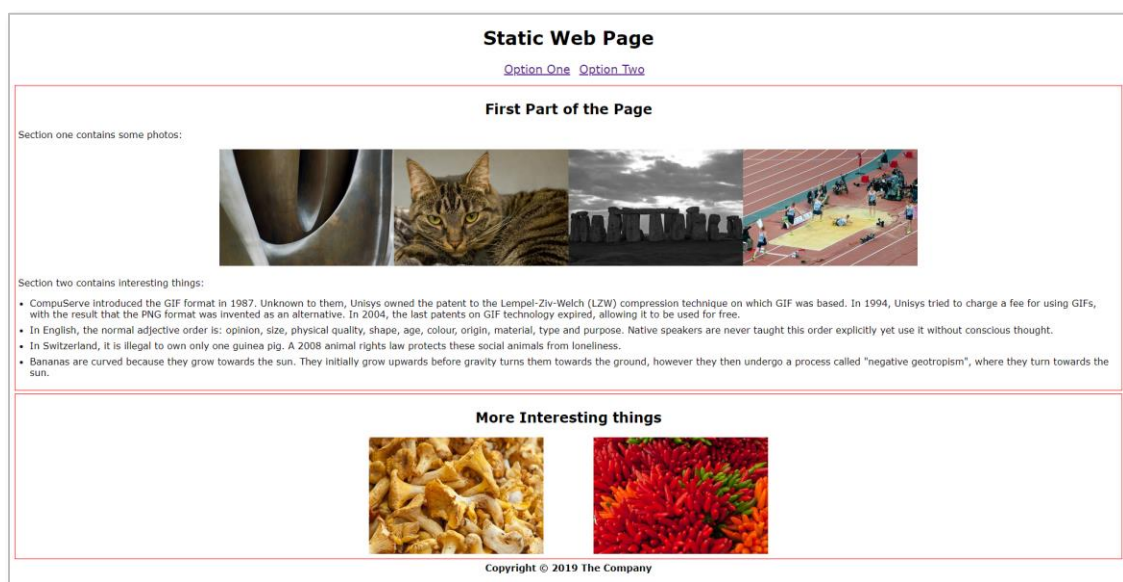
Optional Exercise 3.2: Static Web Pages

This is an optional exercise which can be done if time and interest permits.

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 photos seamlessly.
 - e. The second of the two parts should contain other photos or text. If you use photos, arrange them differently from those in the previous section.
 - f. Optionally, create additional pages for the links and/or clicking the pictures.
3. 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 contain 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 article, blue for section, and green for aside.

Semantic Elements

[Option One](#) [Option Two](#)

First Part of the Page


Section one contains some photos:




Section two contains interesting things:

- CompuServe introduced the GIF format in 1987. Unknown to them, Unisys owned the patent to the Lempel-Ziv-Welch (LZW) compression technique on which GIF was based. In 1994, Unisys tried to charge a fee for using GIFs, with the result that the PNG format was invented as an alternative. In 2004, the last patents on GIF technology expired, allowing it to be used for free.
- In English, the normal adjective order is: opinion, size, physical quality, shape, age, colour, origin, material, type and purpose. Native speakers are never taught this order explicitly yet use it without conscious thought.
- In Switzerland, it is illegal to own only one guinea pig. A 2008 animal rights law protects these social animals from loneliness.
- Bananas are curved because they grow towards the sun. They initially grow upwards before gravity turns them towards the ground, however they then undergo a process called "negative geotropism", where they turn towards the sun.

More Interesting things



Mushrooms



Peppers

Did you know you can put in an aside?

Copyright © 2019 The Company

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.

This page intentionally left blank.

Chapter 4: Client-Side JavaScript Programming

Exercise 4.1: Jumping JavaScript

Time: 10 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?

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 in 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 function and your new function be 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 will randomly select an inspirational message and display it in 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 `addEventListener` 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 you are 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, so 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 useful, 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 then 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, this means 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 also 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/>.
3. Add multiple `<button>` and `<div>` elements to your page.
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 show 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 you are 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, so 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 title of the form 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:

```
npm install
```
3. Start the SimpleServer by typing the following command:

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

Complete the Ajax Client

6. 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 `SELECTS`, review the documentation (e.g., <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select>).
7. Open the file `Ch05\Ajax\ajax_exercise.js` for editing.
8. 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`.

- c. You may wish to log the data returned from the service before attempting to add it to the `SELECT`.
9. Verify that when the web page opens, it sends an Ajax request and creates a `SELECT` using the group list that is returned by the SimpleServer.
10. 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`.
11. Verify that the web page functions as expected.

Bonus Exercise (to be attempted if time permits)

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

Chapter 6: Introduction to Angular

Exercise 6.1: Getting Started with Angular

Time: 30 minutes

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

1. Start Visual Studio Code and open a new folder to act as your Angular workspace folder.
2. Open a command window (also known as Integrated Terminal: `CTRL+`` or `CTRL+'`, depending on your keyboard settings). You can also open a Windows command window and change directory manually.
3. Type the following command and note the output:

```
ng version
```

- a. If the command is not found, you need to install Angular/CLI:

```
npm install -g @angular/cli@latest
```

4. Create a new Angular application by typing the following command:

```
ng new BookStore
```

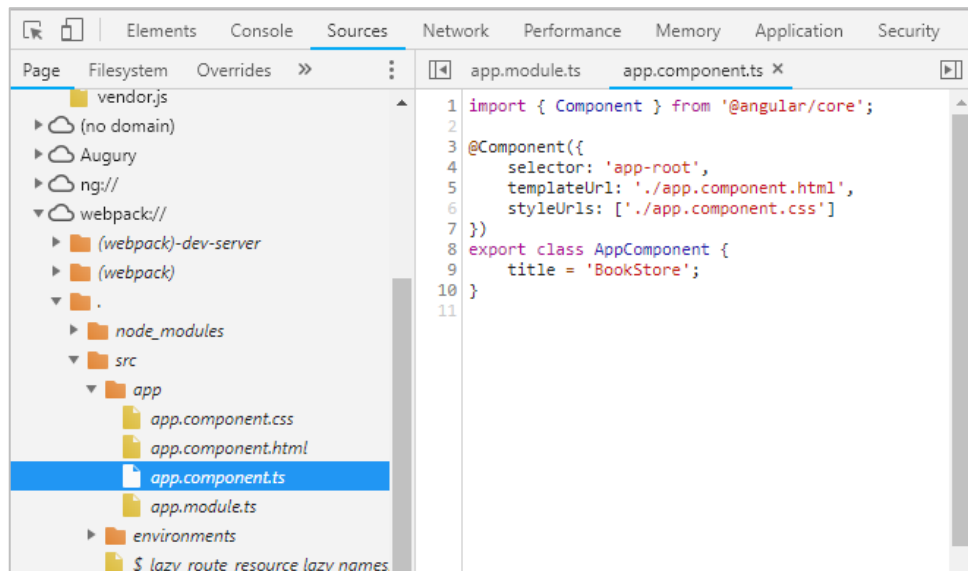
- a. Accept the defaults at each prompt (so, say “no” to adding routing and choose “CSS” for the stylesheet format). Normally you would say “yes” to the first option for anything but the simplest applications, but we will show you how to add routing manually later.
 - b. *Note:* This may take several minutes to complete.
 - c. *Remember:* Patience is a virtue!
5. Explore the folder `BookStore` in Visual Studio Code and examine the file and folder structure.
 6. 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`).

- Make sure the command window is in the application root directory (if necessary, `cd BookStore`) and run the application by typing the following command in the command window:

```
ng serve
```

You can start an Angular application using either `ng serve` or `npm start`. `ng serve` only works with Angular/CLI applications. `npm start` works (or can be made to work) with any application and can also run a script or run commands with preset options (it is particularly useful in that situation).

- Open Chrome and navigate to `http://localhost:4200`. You should (briefly) see the text **Loading...** before `Welcome to BookStore` appears.
- Right-click **Welcome to app!** 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.

- 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`, or whatever you would like to name your bookstore, just as long as it is a different name.
- Save the file.

12. You should now see messages in the terminal/command window as webpack 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.
 - a. *Note:* You may need to click **Play** to debug.
13. 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 Chrome's development tools.
14. Fix the error by replacing the ' character, then take note of the results in both the terminal window and Chrome.

Normally, you can just leave the 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.

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.

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

```
npm install jasmine-node
```
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.
 - a. *Hint:* For an exception that throws a new `Error`:

```
it('should throw error in factorial operation when the number is negative', function () {  
    expect(function () {  
        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. If you have time, add more tests for the basic Math functions in the `MathUtils.js` file.
 - a. Consider both success (positive) and failure (negative) conditions.

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.

In Visual Studio Code, reopen the folder for the Angular BookStore app you created earlier. Or create it now based on Exercise 6.1.

Note: While the only way to really learn how to build an Angular application is to do the work yourself, there are solutions for every exercise. You are certainly welcome to examine the code in any or all of the solutions. Keep in mind that the solutions are simply references and are examples of one way of solving the problems stated in each exercise. Should you decide to run a solution project, you will need to use an integrated terminal in the project directory to run the command `npm install` to install the necessary node_modules. You can then run `npm start` in one terminal window and `npm test` in another terminal window to view the behavior of the solution project.

1. Open a terminal window for the application directory.
 - a. Run the following command to start the application:

```
npm start
```
2. Verify that your app is stable and running by visiting `http://localhost:4200` in your favorite browser.
3. Open `app.component.spec.ts` from the `src/app` folder and examine the contents.
4. This is a very simple test class that worked with the original version of `app.component.ts` and `app.component.html`.
5. Run `npm test` in a second terminal window. This will launch the Karma test runner and run all of the tests defined in `app.component.spec.ts`.
6. 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.

7. The error messages are fairly specific as to the source of the errors.
 - a. If at any time you start to receive very cryptic error messages, consider changing the test script in `package.json` to include the source map option, as shown in the slides.
 - b. Or stop the test (`Ctrl+C`) and then run the following command:

```
npm test --sourceMap=false
```
8. Make the necessary changes to get all the tests to pass.
 - a. If you run into problems, let your instructor know.

Chapter 7: Angular Components

Exercise 7.1: Creating a Component

Time: 30 minutes

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

Open the `BookStore` application directory in Visual Studio Code. Verify that all of the tests run and pass by running `npm test` in a terminal window. Fix if needed. All tests must pass before starting this exercise.

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.

1. 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 and restart it, but most changes should work perfectly.
 - 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. If you prefer, you can periodically stop the compiler to issue `ng` commands and then restart it by typing `ng serve` or `npm start`.
2. 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. Either create a folder called `models` and then a file called `book.ts`.
 - b. Or use the Angular CLI: `ng g class models/book`.

3. 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.

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

4. Correct the error in the `Book` test spec.
 - a. Use a `Book` constructor with 4 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();  
  });  
});
```

5. 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:
 - a. Type the following command to create `BookPageComponent`:

```
ng g component books/BookPage
```
 - b. *Note:* 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.
6. In VS Code, note that a new folder named `books` has been created and another folder, `book-page`, has been created inside it. Examine the files that are contained in the `book-page` folder. These files work together to define the new component.
 - a. Examine the contents of the `book-page.component.html` file
 - b. Eventually we will see this content in the application
7. 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.

8. 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. From the menu that appears, select the option that imports the file with a relative path. 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() {
  }
}
```

9. 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.
10. Open `app.component.html`.
 - a. Remove any extraneous content so the file just contains the greeting.
 - b. Add the Book List at the end by adding:

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

11. If you weren't already, run the application and check that the output is as you expect. It might look a little like this:

Welcome to Angles on Books!

book-page works!

12. 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). Add the same component to the declarations array of the testing module and check that VS Code has automatically added a file import.
 - b. Throughout these exercises, we will suggest some tests, but our suggestions are very far from full TDD. Treat them as inspiration!
13. Now add a placeholder for the book in the component template.
- a. We will tidy this up in the next exercises, but for now just add interpolation bindings for the book's title and book's author.
 - b. Replace the "book-page works" text. It might look something like this:

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

- c. Check that your page displays as you expect.

Exercise 7.2: Unit Testing a Component

Time: 20 minutes

In this exercise, you will create a mock component to ensure good test isolation between the root component and the new book list component. Then, you will write a unit test to check that `BookPageComponent` is working correctly. Finally, you will implement the layout to match the test.

1. 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`). This time, we will add a mock to isolate these two components properly.
 - a. Open `app.component.spec.ts`.
 - b. Remove the file import for the `BookPageComponent`.
 - c. Declare a local mock of `BookPageComponent` that does nothing.
2. Next, we will change the component 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 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: this 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.
 - 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', () => {  
  let fixture = TestBed.createComponent(AppComponent);  
  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');  
});
```

- 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 Chrome, and also check that the tests pass (remember to check that the tests are actually compiling since sometimes structural changes cause the test compilation to crash). To make the table more easily visible, try setting the border to a width of 2 by modifying the `book-page.component.css` file.

```
<table border="2">
  <tr>
    <td>{{book.title}}</td>
    <td>{{book.author}}</td>
  </tr>
</table>
```

Bonus Exercise (to be attempted if time permits)

- 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.
- Improve the look and feel by adding an `<h2>` element above the table with the content `Books`.
- You could also consider defining and using CSS styles to control the styling of the book list.

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 contents of the table 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 so that it now expects 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. Change the `*ngFor` directive to use a tracking method:
 - a. Create a method `trackBook()` inside the `BookPageComponent` class. The method should receive two arguments: a number `i` and a `Book` `book` and return the `bookId` from the method.

```
trackBook(i: number, book: Book): number {  
    return book.bookId;  
}
```
 - b. Next, add a `trackBy` to the `ngFor` directive, assigning `trackBook` as the value.

```
<tr *ngFor="let book of books; trackBy: trackBook">
```
6. Experiment with adding `index`, `first`, and `last` variables to the `*ngFor` and displaying the resulting values inside the table.
7. Try other directives such as `ngClass`, `ngStyle`, and `*ngIf`.
 - 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();
```

Exercise 7.4: Refactoring Components

Time: 30 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`, that 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. Just to be safe, stop the tests by pressing `Ctrl+C` in the integrated terminal that is running the tests.
3. Move any tests for the table contents to the new 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.
 - 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` to be displayed when there are no items in the list, move the mark-up for that as well.

7. Move any styles that apply solely to the book list from `book-form.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. 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);
});
```

- c. 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. At the moment, the book page contains very little apart from the list of books, but that will change over the next exercises.

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`.

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 normally 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 `BookPageComponent` to the exports array of `BooksModule`. This will be the only component exposed outside the module.
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.

Exercise 8.2: Two-Way and Event Binding

Time: 45 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.

Do your lab work yourself and with the help of other students, but if you want to run the solution files in a terminal window, you will need to run `npm install` to add the Angular modules back in. Then you can run `npm start` in one window and `npm test` in another and see how our classroom solutions work. These solutions are just a reference. You need to do the work to get the most out of your hands-on labs.

Important Note: Double-check the directory in a terminal window with `dir` to be sure you are in the right location before running CLI commands.

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.
 - 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 [books]="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`.
 - c. Add the mock `MockBookFormComponent` to the declarations section of the `TestBed.configureTestingModule`.

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

6. We are now going to create 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`.

```
<div>
  <label for="title">Title:</label> <input type="text" id="title" />
  <label for="author">Author:</label> <input type="text" id="author" />
  <button type="button">Add Book</button>
</div>
```

8. Verify the labels, textboxes, and button are above the table of books by viewing the application in your browser.
9. After doing this, it should be clear that the Book Form Component needs a property representing a book to accept the data from the two input fields and a method to be activated by the button.

10. Add the `book` property:
 - 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" />
```

11. 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 add the `imports` array to the decorator because it doesn't exist yet.
12. 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, but it would check that the bindings were declared properly. We will not add any tests to our solution.
13. 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`.
 - 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 really 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.

14. 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 {

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

  constructor() { }

  ngOnInit() {
  }

  add() {
    console.log(this.book);
  }
}
```

15. And `book-form.component.html` should be a little like this:

```
<div>
  <label for="title">Title:</label>
  <input type="text" id="title" [(ngModel)]="book.title" />
  <label for="author">Author:</label>
  <input type="text" id="author" [(ngModel)]="book.author" />
  <button type="button" (click)="add()">Add Book</button>
</div>
```

16. 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.
17. To pass the book object to a handler in the `BookPageComponent`, you will create an `@Output` property and emit an event from the `BookFormComponent`.
- To accomplish this, you will need to import `EventEmitter` and `Output` 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();
});
```

- 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.

18. 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>
```

19. 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 and that could be an issue. The basic test logic is good.

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

21. However, there is a problem.

- a. Add a book to the list.
- b. Now change the contents of the input fields.
- c. What happens?

22. 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);  
}
```

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

- a. In particular, be sure to use some appropriate HTML semantic tags.
- b. Also, define some CSS styles in `book-page.component.css` and use them in `book-page.component.html`.

Bonus Exercise (to be attempted if time permits)

24. Think about the event passing between the form and the list.
- a. Is your test for the `EventEmitter` testing all aspects? What aspect is not being tested? How can you add a test for that?
 - b. Your test is not testing the data passed by the event.
 - c. Is your test for the event handler testing all aspects? Which aspects are not being tested? How could you test them?
 - d. Your test is not testing that the `addBook` method responds to an output event. The issue here is in causing the output event to be triggered. To do this, you will need to add more behavior to your mock `BookFormComponent`. This is similar to what you did to test the input property of the `BookListComponent` mock.

Chapter 9: Pipes

Exercise 9.1: 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 will probably break your tests. 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 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 test because before the transform method is implemented, it will not work, and after the method is implemented, the test will pass. Remember the TDD mantra: Red, Green, Refactor.

5. To make this test pass, you must implement the `NoImagePipe`.
 - a. Implement the `PipeTransform` interface. Change the data type of `value` and the return type to `string`.
 - b. Satisfy the test.
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 in only one unit test.

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 VSC, 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></td>
```

11. Test your application again by viewing `localhost:4200` in your browser.
 - a. This time, you should see the default image.
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;  
  }  
}
```

- c. Alternatively, given the simplicity of this pipe, you could just import the `SharedModule` into the test. This is not the best choice, but it is quick.

This page intentionally left blank.

Chapter 10: Angular Services

Exercise 10.1: 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, return null or 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;
    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. 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
const bookService = jasmine.createSpyObj('BookService',
  ['getBooks']);
bookService.getBooks.and.returnValue(of(testBooks));
```

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

```
TestBed.configureTestingModule({
  declarations: [
    BookPageComponent,
    BookListMockComponent,
    BookFormMockComponent
  ],
  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.

```
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');  
});
```

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

11. You should now have a warm, fuzzy feeling since all of your tests should pass.
12. Let's switch our attention to the `addBook()` method of `BookPageComponent`.
 - a. We do not need the existing test(s) for `addBook` since it/they check that the array is being maintained correctly and that is the responsibility of the service. Instead, we need a test to ensure that the service is being called correctly.

- b. Amend the fake `BookService` to mock `addBook` as well as `getBooks`.
- c. Declare the fake inside the `describe`, but before the `beforeEach`.
- d. The fake should be defined to return its parameter.

```
addBookSpy = bookService.addBook.and(
  .callFake(function (param) {
    return of(param);
  })
);
```

- e. Modify the existing `addBook()` method 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. 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).toHaveBeenCalledBeenCalledWith(expected);
});

it('should respond to the output event from the 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).toHaveBeenCalledBeenCalledWith(expected);
  });
```

13. Verify that everything is still working and all of your tests are passing.
 - a. Well done!

Exercise 10.2: 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 the Tomcat `webapps` folder.
Note: `C:\Apps\apache-tomcat-9.0.36\webapps`.
 - b. Start the Tomcat server by running `bin\startup.bat`.
 - c. Do **not** close the command window.
3. Verify the BookService has deployed correctly by visiting the following URL in your browser:
 - a. `http://localhost:8080/BookService/`

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. Set up the `HttpClientTestingModule`:

```
let httpTestingController: HttpTestingController;

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

- 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 (copy the list of books from the BookService, we will not need it there much longer).

```
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);
    // Assert that there are no outstanding requests.
    httpTestingController.verify();
    // 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 private 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);
    // Respond with empty data.
    req.flush(null);
    // Assert that there are no outstanding requests.
    httpTestingController.verify();
    tick();
  })));
```

13. Now, change `addBook()` to use the `HttpClient` service.
- 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.
 - 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`.
 - 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.
15. Make sure everything is working and all your tests pass.

Exercise 10.3: Handling Errors in a RESTful Service (Optional)

Time: 20 minutes

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

1. Open the `BookService`.
2. Create the error handler:

```
handleError(error: HttpResponse) {  
    if (error.error instanceof ErrorEvent) {  
        // A client-side or network error occurred. Handle it.  
        console.error('An error occurred:', error.error.message);  
    } else {  
        // The backend returned an unsuccessful response code.  
        // The response body may contain clues  
        console.error(  
            `Backend returned code ${error.status}, ` +  
            `body was: ${error.error}`);  
    }  
    // return an observable with a user-facing error message  
    return throwError(  
        'Unable to contact service; please try again later.');  
};
```

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

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

4. 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: HttpResponse;
    let errorReply: string;
    const errorHandlerSpy = spyOn(service,
        'handleError').and.callThrough();
    service.getBooks()
        .subscribe(() => fail('Should not succeed'),
            error => errorReply = error);
    const req = httpTestingController.expectOne(serviceUrl);
    // 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'
    });
    // Assert that there are no outstanding requests.
    httpTestingController.verify();
    // Cause all Observables to complete and check the results
    tick();
    expect(errorReply).toBe(
        'Unable to contact service; please try again later.');
```

5. Check that everything works.
6. Now make the `BookPageComponent` display an error message.
- Add a `DIV` to display it.

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

- Declare a variable: `errorMessage: string;`

- c. Add handling to the subscribe.

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

7. How could you write a test for this functionality? 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();  
});
```

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

Bonus Exercise (to be attempted if time permits)

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

This page intentionally left blank.

Chapter 11: Building an Application

Exercise 11.1: Building an Angular Application

Time: 60 minutes

In this exercise, you will build a new Angular application and view it in the 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 and that will make it easy to mock. However, that would not show any visible feedback. At this stage of experience, there is value in being able to see 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\apachetomcat-9.0.36\webapps`). If Tomcat is not running, start it by running `/bin/startup.bat`.

Verify the service is running by opening `http://localhost:8080/CarService` in any old 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. Create a new directory for your Angular project. Open a terminal window and change the directory to your Angular project directory.
2. Create a new Angular application by typing the following command:

```
ng new ManageCars
```

- a. Take the defaults (this app will not use routing and will use CSS).
- b. **Note:** this may take several minutes to complete.

3. Open the folder `ManageCars` in Visual Studio Code and examine the file and folder structure.
4. Run the application by typing the following command in the command window:
`ng serve`
5. Open Chrome and navigate to `http://localhost:4200`. You should (briefly) see the text **Loading...** before **Hello Angular** appears.
6. First, we will make a small cosmetic change to confirm everything is working.
 - a. Open `app.component.ts` in the `src/app` folder, and change the title to be `Cars World`.
 - b. Change the template, removing all extraneous code.
 - c. Add a copyright centered at the bottom of the page.
 - d. Be sure to use HTML semantic tags in your template code.
 - e. Define a few CSS styles and use them in your template code.
 - f. Check that it displays as expected.
7. Fix the tests.
 - 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 right folder).
 - a. *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, you know what to do!
 - b. *Hint:* You may need to stop the tests (`Ctrl+C`) and then start them again (`npm test`)
 - c. It is highly recommended that you fix the unit tests before moving on to the next step.

Make the CarListComponent

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 using a mock.
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 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 in 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 the service class and initialize it. If you initialize it to different values, you will be able to tell that the right 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 `getCars` method to retrieve the cars from the service.
 - c. Change content of `ngOnInit()` to `this.getCars()`;
 - 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 will need a list of mock cars.

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`.
- 23. Add a private property `URL` to the class, with the value `http://localhost:8080/CarService/jaxrs/cars`, or wherever your service is running.
- 24. Amend the `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 service.
- 26. Fix the tests for the `CarService`.
 - a. The test should use `HttpClientTestingModule`.
 - b. You will need a mock list 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 adds `?filter=price` to the base `url`. (This URL gets a list of the top three cars by price).
28. Add a similar method to `car-list.component.ts` that delegates 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 `getCars()` and `getCarsByPrice()` respectively, e.g., `(click) = "getCars()"`.
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.

Bonus Exercise (to be attempted if time permits)

32. Style the application.
33. Add tests for all the functionality, including at least one negative test for the http service.
34. Look for re-factoring opportunities.

This page intentionally left blank.

Chapter 12: Angular Routing

Exercise 12.1: 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`.
 - a. *Hint:* `ng g component about-page`
 - b. This is a “page” component since it will have its own URL.
 - c. Inside `about-page.component.html`, use semantic tags and some CSS styles to describe your application.
 - d. *Hint:* A footer that is centered is a good place to start.
3. Create the `AppRoutingModule`.
 - a. Using the CLI, create 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`.
 - i. *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. Add styling to make your links look more impressive. Consider styling the active link differently from the inactive one.

Exercise 12.2: 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 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 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 it is not yet doing anything with the route parameter.
8. Now complete the `ReviewListComponent`:
 - a. Add a `private route` parameter of type `ActivatedRoute` to the constructor.
 - b. Inside the `ngOnInit()` method, pass the string `id` to `this.route.snapshot.params[]` and set the return as the value of `this.bookId`.
9. Check the application still works and that the link now passes the `id`.
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 extra test using a mock `ActivatedRoute`.

This page intentionally left blank.

Chapter 13: Angular Forms

Exercise 13.1: 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;
}
```

- 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;

  review: Review;

  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 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. 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. Take a look at your tests and make sure they all pass. They will because we are not testing the form yet!

8. 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()">
```

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

Bonus Exercise (to be attempted if time permits):

10. 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`.

Exercise 13.2: 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. Check the page again 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)">
```

Bonus Exercise (to be attempted if time permits):

8. Use `content.pristine` to ensure the required error message only displays after the user has interacted with the text box. You will need to enter and then delete text in order to see the error message.
9. Enforce a minimum content length using the `minlength` validation attribute.
 - a. Since you are now adding a second validation item, add an outer `div` around the existing validation message and copy the existing `*ngIf` test into the outer `div`. This outer `div` will control whether any error messages appear and the inner `div`s will control the individual error messages, one per error.
 - b. Modify the test in the nested `div` so that it checks `content.errors.required`.
 - c. Then, add a second child of the outer `div` with an appropriate error message for the `minlength` error.

```
<form #revForm="ngForm" (ngSubmit)="submit (revForm)">
  <label>Review:
    <input required minlength="3"
      type="text" name="content" #content="ngModel"
      [(ngModel)]="review.content" />
  </label>
  <button [disabled]="!revForm.form.valid" type="submit">
    Add Review
  </button>
</form>
<div *ngIf="!revForm.form.valid && !revForm.pristine">
  <div class="error" *ngIf="content.errors.required">
    Review must have content
  </div>
  <div class="error" *ngIf="content.errors.minlength">
    Review must be at least 3 characters long
  </div>
</div>
```

10. Add a test for your new validation. It is likely that your old test will now fail (certainly the one above will) since it does not take account of the minimum length requirement. In that case, replace it with a new test.

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

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

    expect(contentControl.status).toEqual('INVALID');
    expect(contentControl.errors).toEqual({ required: true });

    contentControl.setValue('a');

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

    expect(contentControl.status).toEqual('INVALID');
    expect(contentControl.errors).toEqual({
        minlength: {
            requiredLength: 3,
            actualLength: 1
        }
    });

    contentControl.setValue('aaa');

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

    expect(contentControl.status).toEqual('VALID');
});
```

11. Style the error messages and the form (if you didn't do that before).

Exercise 13.3: 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`.
 - 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. The 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` so it 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.

Exercise 13.4: 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 title and author are different, we will need a `FormGroup` that contains a `title` and `author`. (We might normally use the `FormBuilder`, but for now we will just create them directly.)
 - b. The spec will set the two controls to have the same value and check the `FormGroup` has the appropriate error.

```
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).toBeFalse();
    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:

```
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 14: Angular End-to-End (E2) Testing Applications

Exercise 14.1: 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. User adds a book to the list.
2. User chooses the about page.

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

Exercise 14.2: Writing a Simple Protractor Test

Time: 20 minutes

In this exercise, you will write a simple test using protractor. The test will open the application's About Page.

1. Before you start, run the existing E2E test that was created by the CLI.
 - a. If you are still running the live server, you may like to run them using `ng e2e --port 4300`.
 - b. It will fail because we changed the application title.
 - c. Fix the test (`app.e2e-spec.ts`) and run it again.
2. Now 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.
 - c. If you did this option in the previous exercise, you may have the information already.
3. Write the test code in a new spec file, `about.e2e-spec.ts`.
 - 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, but the details will differ:

```
it('should navigate to about page', () => {  
  appPage.navigateTo();  
  appPage.getAboutLink().click();  
  expect(aboutPage.getAboutText())  
    .toContain('This is the Angles on Books  
application');  
});
```


4. Create the two-page objects to support this spec.
 - a. The app page (`app.po.ts`) may look like this (change the selectors to suit your application):

```
import { browser, by, element } from 'protractor';

export class AppPage {
  navigateTo() {
    return browser.get('/');
  }

  getTitleText() {
    return element(by.css('app-root h1')).getText();
  }

  getAboutLink() {
    return element(by
      .cssContainingText('app-root a', 'About'));
  }
}
```

- b. The about page (`about.po.ts`) may look like this (again change the selectors to suit your application):

```
import { by, element } from 'protractor';

export class AboutPage {
  getAboutText() {
    return element(by
      .css('app-about-page p')).getText();
  }
}
```

- c. The setup code for the spec will look a little like this:

```
let appPage: AppPage;
let aboutPage: AboutPage;

beforeEach(() => {
  appPage = new AppPage();
  aboutPage = new AboutPage();
});
```

5. Run the tests.
 - a. Work on them until they pass.

Exercise 14.3: Entering Data in E2E Tests

Time: 30 minutes

In this exercise, you will write a protractor 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?
 - d. If you did this option in the previous exercise, you may have the information already.
2. Write the test code in a new spec file, `book.e2e-spec.ts`.
 - 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 will differ:

```
it('should add a book', () => {  
    const title = 'Neuromancer';  
    appPage.navigateTo();  
    expect(bookPage.bookExists(title)).toBe(false);  
    bookPage.getTitleControl().sendKeys(title);  
    bookPage.getAuthorControl().sendKeys('William Gibson');  
    bookPage.clickAddBook();  
    expect(bookPage.bookExists(title)).toBe(true);  
});
```

h. And your page object may look like this:

```
export class BookPage {
  bookExists(title: string) {
    return element(by.linkText(title))
      .isPresent();
  }

  getTitleControl() {
    return element(by.css('app-book-form
input#title'));
  }

  getAuthorControl() {
    return element(by
      .css('app-book-form input#author'));
  }

  clickAddBook() {
    return element(by.buttonText('Add Book')).click();
  }
}
```

3. If your test doesn't work the first time, you may need to stop and restart the service to re-run the test, since otherwise the book may already be present.
- This is rather inconvenient and may compromise our ability to use this test systematically. Let's change the test so that it doesn't use a fixed value.
 - There are libraries available that generate random "realistic" string data, but for our purposes, we will use `uuid`, which is already installed.
 - Add an import: `import { v4 as uuid } from 'uuid';`
 - And replace the fixed title with a call to `uuid()`:

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

- 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. Refactor the page object so it has an `addBook()` method:

```
export class BookPage {
  bookExists(title: string) {
    return element(by.linkText(title))
      .isPresent();
  }

  getFormControl(id: string) {
    return element(by.css(`app-book-form input#${id}`));
  }

  getTitleControl() {
    return this.getFormControl('title');
  }

  getAuthorControl() {
    return this.getFormControl('author');
  }

  clickAddBook() {
    return element(by.buttonText('Add Book')).click();
  }

  addBook(title: string, author: string) {
    this.getTitleControl().sendKeys(title);
    this.getAuthorControl().sendKeys(author);
    this.clickAddBook();
  }
}
```

- b. Change the test to use this method.

Exercise 14.4: Writing an E2E Test for ManageCars (Optional)

Time: 30 minutes

In this exercise, you will write a protractor E2E test for your ManageCars application (from Exercise 11.1).

The test should start from the application home page, click the button to get a list of cars by price and check that the right cars are displayed.

1. Start by taking the actions yourself. Use developer tools where appropriate.
 - a. How will you find the button?
 - b. Once the button has been clicked, how will you know that the page has updated correctly?
2. Implement the test actions.
 - a. Use a Page Object to isolate your test from the page structure.

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 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 from the BookService. The URL for the application and the URL for the service will both start `http://localhost:8080/BookService`.
 - a. The application will be served from that URL.
 - b. The service will be served from that URL suffixed with `/jaxrs`.
2. Enable the live server proxy.
 - a. Create a file `proxy.conf.js` 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
  }
}
```

- b. Edit the `angular.json` file and add a `proxyConfig` setting pointing to the new config file.
3. Edit the `BookService` to use a different URL. Also, change the expected URL in the `BookService` tests.

```
private url = '/BookService/jaxrs/books';
```

4. At this point, the application should run and all the tests should pass.
 - a. You will need to stop and restart the live server to have the proxy setting take effect.
5. Build the application.

```
ng build
```


6. Examine the `dist` sub-directory.
 - a. Open one of the files that start `main` in an editor. 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.
7. Build the application for production.

```
ng build --prod --base-href=/BookService/
```
8. Examine the `dist` sub-directory again.
 - a. Open one of the files that start `main` again. Can you find any of your components in it?
 - b. Compare the file names and sizes with your previous list.
9. You can deploy the Angular application to the Tomcat server. You will need all the files in the `dist/BookStore` subdirectory.
 - a. If you are running the server in Eclipse, copy all the files to the `webapp` folder (under Deployed Resources). Clean the project, build and deploy to the test server.
 - b. If you are running the server through the standalone Tomcat server, copy all the files directly to the `webapps/BookService` folder, stop and restart the server. You should not normally do this, but it will achieve the same objective for now.
10. The application should now be available at `localhost:8080/BookService`.
 - a. Note that it will not run in the Eclipse built-in browser.
 - b. The `noImage` pipe will not serve the right file. The address for the image should be relative to the base href, but we sent it relative to `"/`.
11. If you are running through Eclipse, create a new server on port 8088 and run the application there. You should see that everything still works, proving that it is now self-contained.

Bonus Exercise (to be attempted if time permits)

12. Fix `NoImagePipe`. It would be great if we could use a relative path, but the assets folder is deployed differently in the built system, making that impossible.

- a. Get the base href (notice we have removed the leading `/` from the string literal):

```
constructor(private locationStrategy: LocationStrategy) {}

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

- b. Create a `LocationStrategy` for the tests:

```
let locationStrategy: LocationStrategy;

beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [
      RouterTestingModule
    ]
  });
  locationStrategy = TestBed.inject(LocationStrategy);
});
```

- c. Fix the tests by adding `locationStrategy` to the construction of the `NoImagePipe`. For example:

```
it('create an instance', () => {
  const pipe = new NoImagePipe(locationStrategy);
  expect(pipe).toBeTruthy();
});
```

13. It was also tedious and unreliable to have to update the service URL in both the `BookService` and the tests. 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.prod.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. Do the same for the tests.
- ```
private url = environment.serviceUrl;
```
15. Check that the tests and the application still work.
- a. Try re-deploying the application to whichever Tomcat server you are using.

This page intentionally left blank.

## 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. It is the directive's job 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.

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

```

```

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.
- 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 { }
```

- 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.
- Create a new folder under `src/app`, call it `mocks`.
- 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 `async` pipe. The `trackBy` must occur *after* the `async`.
  - 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(serviceUrl + '?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);

 // Finally, assert that there are no outstanding requests.
 httpTestingController.verify();
 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"
 [nobeoks]="nobeoks"></app-book-list>
```

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

```
<div id="nobeoks" *ngIf="nobeoks">
 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 `nobeoks` instead of `books`.
17. There should be a test of `BookPageComponent` checking that it sets `nobeoks` 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 `nobeoks` 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.nobeoks).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(async(() => {
 // 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,
 BookListMockComponent,
 BookFormMockComponent
],
 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.
- It used to be there, but because we are not doing integration testing, we didn't notice that it was effectively disabled.
  - 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.
- 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));
}
```

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

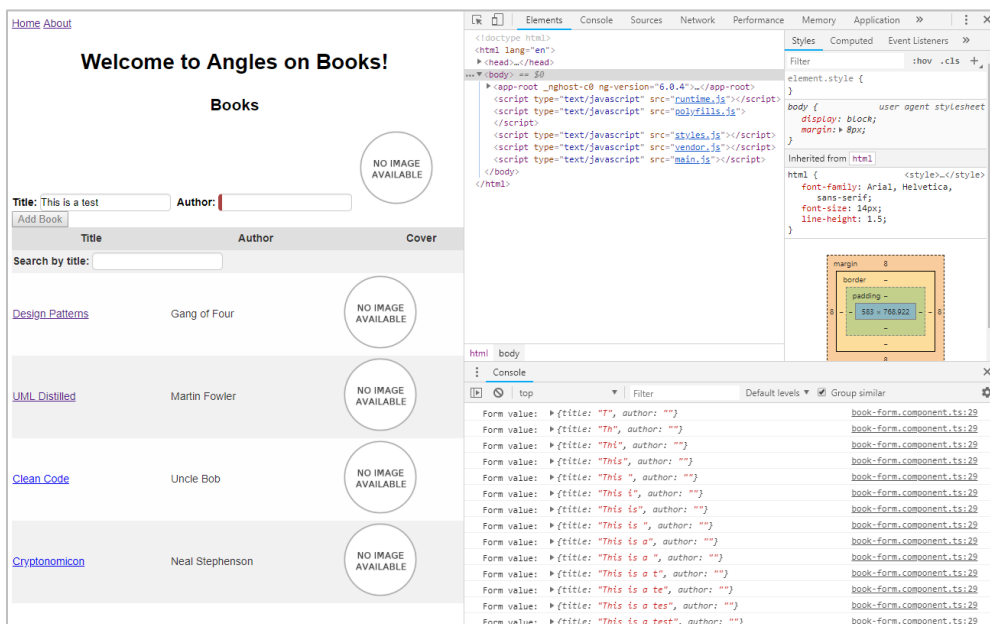
1. You will now change the book form so that it responds to key presses. You could do validation, but for test purposes we will just convert the string to title case.
  2. Open `BookFormComponent`.
    - a. Add a property `sub` (of type `Subscription`) to the class.
    - b. At the end of the `ngOnInit()`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



The screenshot shows a web application titled "Welcome to Angles on Books!". It features a "Books" section with a form to add a new book. The form has two input fields: "Title: This is a test" and "Author: ". Below the form is a table listing books. The table has columns for Title, Author, and Cover. The books listed are "Design Patterns" by Gang of Four, "UML Distilled" by Martin Fowler, "Clean Code" by Uncle Bob, and "Cryptonomicon" by Neal Stephenson. Each book has a placeholder for a cover image labeled "NO IMAGE AVAILABLE".

The right side of the screenshot shows the browser's developer tools. The "Console" tab is active, displaying a series of log messages: "Form value: {title: 'T', author: ''}", "Form value: {title: 'Th', author: ''}", "Form value: {title: 'Thi', author: ''}", "Form value: {title: 'This', author: ''}", "Form value: {title: 'This ', author: ''}", "Form value: {title: 'This t', author: ''}", "Form value: {title: 'This is', author: ''}", "Form value: {title: 'This is ', author: ''}", "Form value: {title: 'This is a', author: ''}", "Form value: {title: 'This is a t', author: ''}", "Form value: {title: 'This is a te', author: ''}", "Form value: {title: 'This is a tes', author: ''}", and "Form value: {title: 'This is a test', author: ''}".

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.