Testing Angular Applications

Jesse Palmer
Corinna Cohn
Craig T. Nishina
Mike Giambalvo

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Testing Angular Applications
Version 10

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Testing Angular Applications: Covers Angular 2*. Writing this book has been a great experience, and I'm excited to make this book a valuable resource when it is released.

A lot of time and effort has been put into making the content of the book as approachable as possible for all readers. The goal is to provide technical details that lay a foundation to build on when it comes to testing Angular Applications.

This is an intermediate level book for current web developers that want to learn how to test Angular applications. Even though this book is being designed for web developers who have knowledge of JavaScript and web development, it does not assume that the reader has prior experience with testing or Angular.

The first part of the book will give you an overview of what testing Angular applications is all about. Immediately after the introductory chapter, you will start writing tests using the Jasmine and Karma frameworks. After you get your feet wet by testing out classes, you will then learn how to test components which are the fundamental building block in Angular.

In the second part, you will we dive into more unit testing the other major concepts in Angular, which are directives, pipes, services, and routing. In the final part of the book, we will explore testing user interactions through the browser by creating end-to-end testing with the Protractor framework.

While you are reading through the book, I hope you'll post to the Author Online forum. Your honest and open feedback is valuable and will help make this book of better service to others in the Angular community. I look forward to seeing your comments, and I will be responding to them throughout the process.

— Jesse Palmer

brief contents

1

Introduction to testing Angular applications

This chapter covers:

- Understanding Angular testing
- Getting a first look at TypeScript
- Understanding the basics of unit and end-to-end tests
- Introducing Jasmine, Karma, and Protractor

Poorly written code, buggy functionality, and bad refactoring practices are can lead to unreliable applications. Writing good tests will help prevent situations like these from negatively affecting your application. Therefore, it is vital that you have an application that has adequate test coverage to create a sustainable application that may need to be supported for many years. We write tests that help guard against breaking application functionality when you add new features or make bug fixes.

If you have developed an Angular application, you may know that it is a great framework for building testable web and mobile web applications. That is because one of the goals when Angular was written was to be a testable framework.

Even though testing Angular applications is of upmost importance, in the past figuring out how to test Angular applications has been challenging. You may be able to find a blog post or two, perhaps a video, but generally there is a lack of materials to help guide you through all the different aspects of testing in one place together. Well you are in luck! In your hands (or on your screen) you have everything you need to get started with testing Angular applications.

This book will help you build a foundation so that you will be able to test the most important parts of Angular applications with confidence. I will assume that you have some

familiarity with the Angular framework, TypeScript, and command-line tools. If you haven't written a test, this book will teach you enough fundamentals to get you started.

If you do not have experience with Angular though, now is a great time to learn about the Angular applications. The framework is matured and is fully ready to be used for applications. For newbies, I would encourage you to walk through the tutorials and introductory information you can find at angular.io.

In this first chapter, you are going to get an overview of testing Angular applications, take a brief look at TypeScript, learn about the testing tools that you will be using, and be introduced to unit and end-to-end tests. Let's get started!

1.1 Angular testing overview

Most the Angular testing you will find in the world include two types of tests: Unit Tests and End-to-End Tests (E2E). Therefore, the bulk of this book will revolve around those two types of tests.

This book is separated into two parts. The first part of this book covers unit testing, where you will learn how to test units of code. You will learn how to create unit tests for Components, Directives, Pipes, Services, and Routing—using testing tools like the Karma and Jasmine—and executing those tests using the Angular CLI. Table 1.1 is a breakdown of each the testable concepts we will cover in the first part of the book.

Table 1.1: Description of key Angular testable concepts

Concept	Description
Components	Components are chunks of code that can be used to encapsulate certain functionally that can be then reused throughout the application. Components are types of directives (see below) except they include a view or HTML template.
Directives	Directives are used to manipulate element that exist in the DOM or could also add or remove elements to the DOM. Examples of directives that come included with Angular are `ngFor`, `ngIf`, and `ngShow`.
Pipes	You will use Pipes to transform data. For example, say you want to turn an integer into currency. You would use a currency filter to turn 1500 into $15.00.
Services	Even though services do not technically exist in Angular the concept is still very important. You will use services to fetch data and in turn inject that data into our components.
Routing	Routes allow users to navigate from one view to the next as they perform tasks in the web application.

In the second part of the book, you will dive into end-to-end testing using the Protractor framework. You will get practice writing tests as if the interactions where coming from the user in a browser.

As for which version of Angular you will be using, this book is written to be compatible with versions of Angular 2 and later. The rational is because Angular 2 was a complete rewrite from AngularJS 1.x.

> **IT'S JUST ANGULAR** In the past people have referred to Angular as AngularJS, Angular 1, Angular 2, Angular 4, etc. From here on out AngularJS will be used for Angular 1.x and Angular will be used for versions 2 and above. To read more about the decision, checkout this blog post: http://angularjs.blogspot.com/2016/12/ok-let-me-explain-its-going-to-be.html.

In the next section, we are going to look at TypeScript which is the language you will be using to write tests in this book.

1.2 Getting friendly with TypeScript

TypeScript, is an open-source language created by Anders Hejlsberg at Microsoft in 2012, who is also the creator of C#.[1] The major problem that Andres attempted to solve with TypeScript is that JavaScript was never meant to be used with large-scale applications. The first version of JavaScript was created in 1995 in 10 days by Brenden Eich[2] and was meant to be used as scripting language for adding interactivity with web pages.

Even though you can build Angular applications with native JavaScript, it is recommended that you use TypeScript because most of the examples, tutorials, documentation, code examples, etc. will use TypeScript. In fact, the Angular framework itself is built with TypeScript.

The benefits of using TypeScript is that that TypeScript adds annotations, static typing, and classical object-oriented features like interfaces and code encapsulation that are needed for enterprise applications, but still utilizes the key features of JavaScript.

You will find that the syntax of TypeScript is very much like JavaScript. TypeScript is actually a superset of JavaScript. In figure 1.1, you will find a simple graphic, that displays that TypeScript key features (outer circle) encompassing the ES6 version of JavaScript key features (inner circle).

[1] TypeScript." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc., 17 December 2017. Web. 17 December 2017 <https://en.wikipedia.org/wiki/TypeScript>

[2] "JavaScript." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc., 17 December 2017. Web. 17 December 2017 <https://en.wikipedia.org/wiki/JavaScript#Beginnings_at_Netscape>

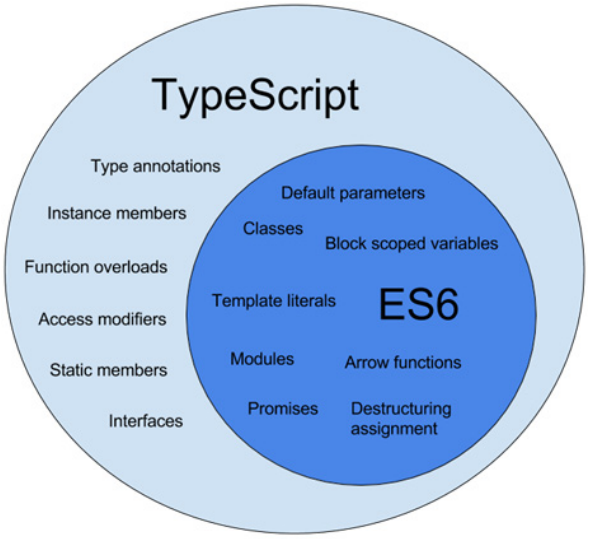

Figure 1.1 Schematic diagram to illustrate that TypeScript is a superset of JavaScript

TypeScript includes most of the key ES6 features so just about any valid ES6 or ES5 features is perfectly valid TypeScript. In other words, you could have a perfectly valid TypeScript file that is just plain, old, vanilla JavaScript.

TypeScript compiles to JavaScript. The good news is that the Angular CLI will compile TypeScript automatically out-of-the-box. If you are familiar with JavaScript and have some knowledge of object-oriented programming, I have no doubts that you will be able to pick it up quickly. If you are new to object-oriented programming, then that is perfectly fine. It just may take a little bit more work to get you up to speed.

In listing 1.1, we create a simple class called `Cat` that demonstrates the what a class looks like in TypeScript.

Listing 1.1 Simple TypeScript Class Example

```
export class Cat {
    private _name: string = '';     ❶ & ❷ & ❸

    constructor(name? : string) {   ❶ & ❹
        this._name = name;
    }

    get name() : string {           ❶
        return this._name;
    }

    set name(name : string) {       ❶
        this._name = name;
    }
```

```
    toString() : string { ❶
        return `This cat's name is ${this._name}!`;
    }
}

const cat = new Cat('Nicky');
```

❶ "string" types are added to specify return, variable, and parameter types.
❷ Private member.
❸ _name is set to an empty string by default.
❹ The '?' makes parameters optional.

There are several noticeable concepts even in this simple example. The `class`, `get`, `set`, and `constructor` keywords along with template string found in the `toString()` method are all part of the ECMAScript 6 specification, so those keywords are not exclusive to TypeScript.

The features in the example that belong to TypeScript are the actual variable types, the `private` access modifier, and the "`?`" operator which allows for an optional parameter.

In the `Cat` class, you will notice that keyword `string` is sprinkled throughout the example. This is where we are specifying the expected types. The ability to specify types is the key advantage of using TypeScript, hence the name.

We are going to cover TypeScript throughout the book, but if you want to get your feet wet with TypeScript then visit http://www.typescriptlang.org/. The website contains a Playground feature there where you can write and test code.

Now that we have gained some knowledge about TypeScript, let's look at the different types of tests that we will be writing in this book.

1.3 Types of tests

In this book, we will be covering two types of tests: unit tests and end-to-end (E2E) tests. These are the two types of tests you are going to see in the wild for testing Angular applications. In this section, we are going to take a brief look at those types of tests.

1.3.1 Unit Tests

Unit tests are written to test the functionality of basic parts or units of code. Each unit test should only test a single responsibility of the source code. We can test functions, methods, objects, types, values, and more all with unit tests. The advantages of using unit test is that they tend to be fast, reliable, and repeatable if the are written correctly and run in a proper environment.

The name of the framework that we will be using to write our unit tests is called Jasmine. Jasmine is a behavior-driven development framework for testing JavaScript code. Listing 1.2 shows the code for a very basic unit test written in using the Jasmine framework.

Listing 1.2 Example of Simple Unit Test

```
describe('super basic test', () => {
       it('true is true', () => {
         expect(true).toEqual(true); (1)
       });
});
```

(1) **Smoke check to see if `true` equals `true`.**

All we are doing in listing 1.2 is checking to see that the Boolean value `true` is equal to the Boolean value `true`. This test serves as primary as a smoke check and nothing more. A *smoke check* is a test to see if all the parts of your testing environment are set up correctly and you are just attempting to get a test to pass. We would not want to add a test this simple to a production application.

An example of a better unit test is if had a class written and you wanted to test out the getters and setters where updating an instance variable like name, then test for the getters and setter would be appropriate.

In listing 1.3 we write a slightly more sophisticated unit test that tests out the `Cat` class that we created in our earlier example found in listing 1.2.

Listing 1.3 Better example of a Unit Test

```
import { Cat } from './cat';

describe('Test Cat getters and setters.', () => {
    it('The cat name should be Gracie', () => {
        const cat = new Cat();
        cat.name = 'Gracie';
        expect(cat.name).toEqual('Gracie'); (1)
    });
});
```

(1) **Checks that the cat name is as we expected.**

Even though unit tests tend to be reliable, they are not the best type of test for reproducing real user interactions.

1.3.2 End-to-end tests

End-to-end tests (E2Es) are used to test the functionality of an application by simulating the behavior of an end user. For example, we may have an end-to-end test that tests that a modal correctly appears after a form is submitted or a page renders certain elements on page load, such as buttons or text.

End-to-end tests do a very good job with testing the application for an end user's standpoint, but the performance of end-to-end tests can be slow. Due to the fact that end-to-end tests can be slow, E2Es can be the source of false positive failing tests due to timing out

issues. Because E2Es are prone to timing issues, it is generally preferred to write unit tests in lieu of E2Es whenever possible.

For writing our E2E tests in this book, we will be using the Protractor E2E test framework developed by the Angular team. The main benefit of using Protractor is that it allows you to test your application automatically as an end user would in the browser.

Listing 1.4 contains a sample end-to-end test written using Protractor that checks the sample project's website and ensures that the title of the page is equal to "Contacts App Starter".

Listing 1.4 Example of Simple End-to-End

```
import { browser } from 'protractor';

describe('Contacts App title test', () => { (1)
  it('Title should be correct', () => { (2)
    const appUrl = 'https://contacts-app-starter.firebaseapp.com/';
    const expectedTitle = 'Contacts App Starter';
    browser.get(appUrl);
    browser.getTitle().then((actualTitle) => {
      expect(actualTitle).toEqual(expectedTitle); (3)
    });
  });
});
```

(1) The suite of test that we want to run. In this case, the test involves the title of the test app we will be using.
(2) The logic of the test.
(3) The specific test case that we are trying to prove as true.

We will be exploring unit tests and end-to-end tests in much greater detail throughout the book. If you understand these two types of tests, you will understand much of Angular testing.

1.3.3 Unit tests versus end-to-end tests

Both unit tests and end-to-end test come with their own set of advantages. We have discussed these advantages a little so far, but in table 1.2, we summarize the pros and cons for the different types of tests.

Table 1.2: Advantages in using unit tests vs end-to-end tests

Feature	Unit Test	End-to-End Test
Speed		
Reliability		
Helps enforce code quality		

Cost effective		
Mimics user interactions		

Let's take discuss each one of these features one by one.

SPEED

Since unit tests operate small chucks a code they can run very quickly. Because end-to-end tests rely on testing through a browser they tend to be slower.

RELIABILITY

Since there tends to be more dependencies and complex interactions associated with end-to-end tests, they can flaky which will lead to false positives. Executing unit tests rarely results in false positive. If a well written unit test fails you can trust that there is a problem with the code.

HELPS ENFORCE CODE QUALITY

You may find that a test is tricky to write for a method. Refactoring the code so that is smaller and only has a single responsibility will make the method easier to test. Plus, this will help you improve your code quality.

COST EFFECTIVE

Because end-to-end tests take longer to execute and can fail at random times, there is a cost value associated with that time. It can also take longer to write end-to-end tests because they may build on other complex interaction that can fail themselves. So, there is a real developer cost as well when it comes to writing end-to-end tests.

MIMICS USER INTERACTIONS

Mimicking user interactions with the UI is where end-to-end tests shine. Using Protractor, we can write and execute test as if it was a real user interacting with the user interface. You can simulate user interactions using unit tests, but it may be easier to write the tests as an end-to-end test since that is what they are made for.

Both types of tests are important to have to thoroughly test your applications. You can test a lot of the functionality that a user would perform just by writing unit tests. Though for important functionality should be tested with an end-to-end test.

Generally, you want to have more unit tests than end-to-end tests in your project. A software developer named Mike Cohn created a popular way of visualizing the breakdown of

allocation of the different tests called the testing pyramid. In figure 1.2, you can see a common percentage allotment of the different tests you should include in your project.

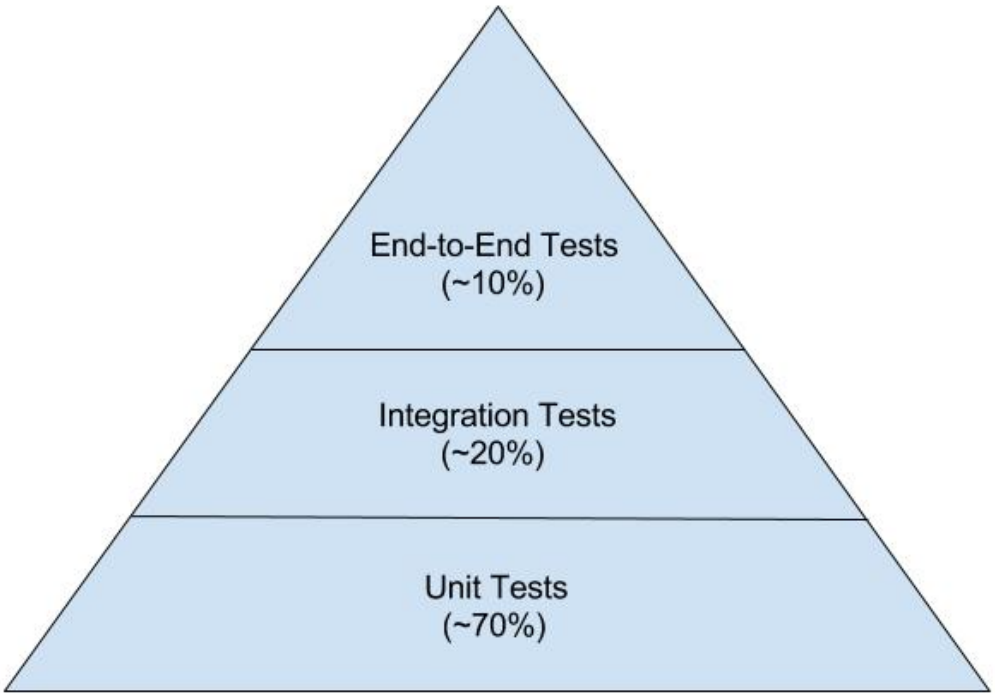

Figure 1.2 Testing pyramid of the recommended distribution of the different types of tests in your project

You may have noticed that the pyramid includes integration tests. Integration tests are used to test that a whole system works correctly. For our purposes, we will roll the integration tests into end-to-end tests. Since end-to-end tests the whole UI system.

So now you should understand the basics of Angular testing, in the next chapter we will dive into writing our first real tests.

1.4 Summary

In this chapter, you learned the following:

- **Angular:** Angular provides a great framework for building testable web and mobile applications. Angular is based heavily on the concept of components, which are like building blocks for developing applications.
- **Types of Tests**: There are two types of tests that you will run across when developing Angular applications. These two types of tests are unit tests and end-to-end tests. Unit

tests are used to test out basic parts or units of code, while end-to-end tests are written in a way that they simulate user interactions.

- **TypeScript, the language of Angular**: TypeScript is a language that is a superset of JavaScript, created by Microsoft. Angular is built with TypeScript and while developers can use native versions of JavaScript (ES5 or ES6) or compile-to-JavaScript languages like CoffeeScript or Dart, TypeScript is recommended. Most of the coding examples, tutorials, blog posts are going to be using TypeScript, so it is in your best interest to dive in and start using TypeScript.
- Plus, TypeScript comes with some nifty features, such as, the ability to assign types, declare private variables, and use object-oriented features, like interfaces, not found in native JavaScript.
- **Testing Tools**: Protractor, Jasmine, and Karma are the primary tools that make the testing of Angular applications a cinch. They are tools that are written in JavaScript and we will configure them to run automatically while we write our tests and during our build process.

Now it's time to roll up our sleeves because in the next chapter we will get down to work by writing our first Angular test.

2

Creating your first tests

This chapter covers:

- **Writing basic unit tests using Jasmine**
- **Using `beforeEach`, `afterEach`, `it`, `describe`, and matcher functions**
- **Testing classes**

Now that we understand the basics about Angular testing from the previous chapter, in this chapter we will get started writing actual tests. Before you go to the next section, follow the project-setup instructions in appendix A to install all necessary tools, the Angular CLI, Jasmine, Protractor, and Karma. You'll be using the sample project throughout the book.

After we get the sample project up and running, we'll begin writing basic tests by using Jasmine, the behavior-driven testing JavaScript framework that we will be using throughout the book. We won't be using any of the testing modules that come included with Angular in this chapter. We will save those for the next chapters.

The reason why we want to cover this is because there are going to times when you want to test something and you do not need help from Angular. There could be a Pipe that transforms dates or there could be a function that performs a calculation that need to be tested. You could test these without any help from the Angular test modules.

In this chapter, we will be testing classes without the help of the Angular framework itself. If you feel comfortable with the fundamentals of writing basic unit tests you may want to skip this section and start in chapter 3, Testing components. Otherwise, let's get started!

2.1 Writing tests using Jasmine

As we first discussed in chapter 1, Jasmine is a behavior-driven development (BDD) framework that is a popular choice when testing JavaScript applications. BDD is a great

methodology to use because you can use it to explain the why of things. The advantages of writing tests using BDD is that the test code that we will be writing will read close to English, as you will see in our examples.

Though you can write Jasmine tests in JavaScript, we will be writing all our tests (and all code in this book) using TypeScript to maintain consistency. Because these are built-in functions in Angular, you can use them without importing the Jasmine library.

2.1.1 Writing basic tests

To write our first test, we will create a smoke test similar to the one we looked at in chapter 1.

Jasmine functions

Jasmine provides several built-in functions, `describe`, `it`, and `expect`, which you will see frequently throughout this book. Let's look at these functions.

DESCRIBE

The `describe` function is used to group together a series of tests. This group of tests is known as a *suite*. The `describe` function takes two parameters, a string and a callback function, in the following format:

```
describe(message describing the test suite, describe callback);
```

You can nest as many `describe` functions as you want to organize your tests. You can have as many `describe` functions as you want. The number of `describe` functions depends on how you want to organize your tests into suites.

IT

The `it` function is used when we want to create a specific test. Like the `describe` function, the `it` function takes two parameters, a string and a callback function. And also like the `describe` function, we can store executable code in the `it` function. The `it` function can be used to separate each test and uses the following format:

```
it(message describing the test, it callback);
```

EXPECT

The `expect` function is where you want to write the code for the test to work. These lines of code are also know as assertion, because you are asserting something as being true. In Jasmine, the assertion is in two parts. The `expect` and the matcher. The `expect` function is where you pass in the actual value, for example, a Boolean value `true`. The second part of the assertion is the *matcher* function, which is where we put the expected value. Matcher functions examples include `toBe()`, `toContain()`, and `toThrow()`, `toEqual()`, `toBeTruthy()`, `toBeNull()`, and more. There are many matchers you can use in Jasmine by default. To find out more information about matchers, checkout the Jasmine documentation at https://jasmine.github.io/.

Keep in mind when you are writing your assertions, that a rule of thumb is you want to try to only have one assertion per test. Sometimes, you will find it usefully to have multiple assertions. In those cases, each assertion must be true for the test to pass.

We'll use the `describe` function to group tests together into a test suite and use the `it` function to separate individual tests. All we are doing in our first test is checking to see that the Boolean value `true` is equal to the Boolean value `true`. The expect function is where we want to prove our assertion and it will be chained together with a matcher, which in this case is `toBe(true)`.

This test serves primarily as a sanity check and nothing more. A sanity check is a test to see if all the parts of your testing environment are set up correctly. You are just attempting to get a test to pass.

Navigate to `website/src/app` and create a file named `first-jasmine-test.spec.ts`. We will use the following naming convention for all our unit tests:

```
<name of file tested>.spec.ts
```

Since this is a one off test and we aren't testing a file, we will give this test the generic, `first-jasmine-test.spec.ts` name. We will be using the actual file name that we are tested in the future. You may have noticed that the word `spec` is included in the filename. If you are curious as to why we have `spec` in the filename it is because `spec` stands for specifying. This means that the test is verifying that a specific part of the codebase works as described in the test file. Generally, you will see this format used for unit tests with Angular.

Inside the `first-jasmine-test.spec.ts` add the following code in listing 2.1.

Listing 2.1 First Jasmine test

```
describe('Chapter 2 tests', () => {          (1)
    it('Our first Jasmine test', () => {     (2)
        expect(true).toBe(true);             (3)
    });
});
```

(1) **Group tests into a test suite**
(2) **Separate individual tests**
(3) **Assertion (true) and matcher toBe()**

To execute our tests, will use the Angular CLI command `ng test`. Using `ng test` any file that is named with the proper format. As long as the file ends in `.spec.ts` the test will be executed. Now, go ahead and run `ng test` in your terminal window you have using for navigation.

A browser should open automatically and you should see something similar to figure 2.1.

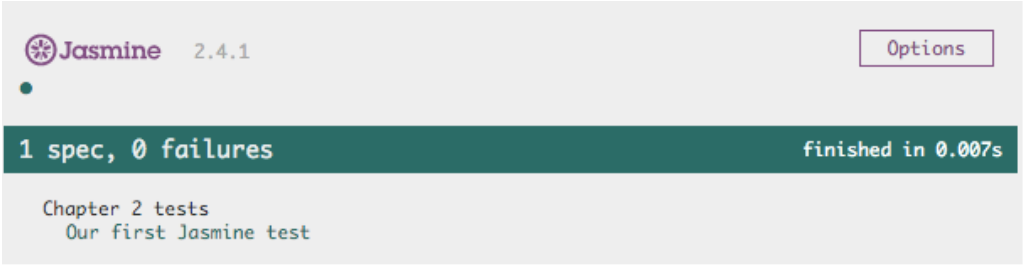

Figure 2.1: Our first passing test

Now that our first is test passing, let's try an exercise to practice what we just learned.

EXERCISE

Write a test that proves that 2 + 2 equals 4 inside your previous `describe` function, but in a new `it` function.

SOLUTION

Now you should have two basic tests. Your test should look similar to the code in bold in listing 2.2.

Listing 2.2 Second basic test

```
describe('Chapter 2 tests', () => {
       it('Our first Jasmine test', () => {
         expect(true).toBe(true);
       });

       it('2 + 2 equals 4', () => { (1)
         expect(2 + 2).toBe(4);
       });
});
```

(1) Second test

Run `ng test`. When the browser window opens, you should see something like the image in figure 2.2.

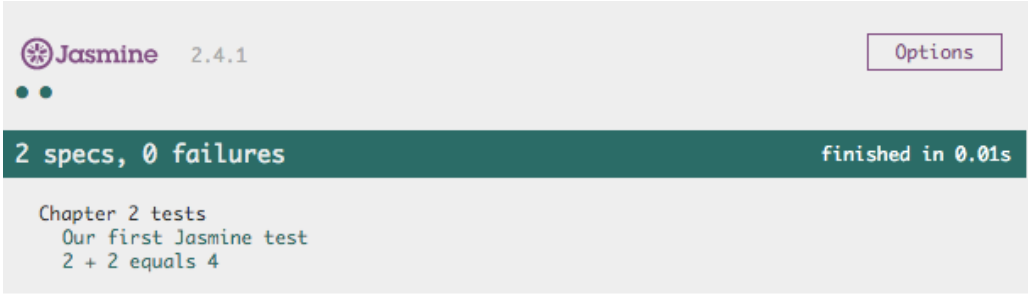

Figure 2.2: Our second passing test

Now that you have two passing tests, let's see what happens when you have a failing test. Go ahead and change your assertion in the `toBe` part of the assertion to something other than `4`. After you change the value run `ng test` and you should see something like the screenshot in figure 2.3 below.

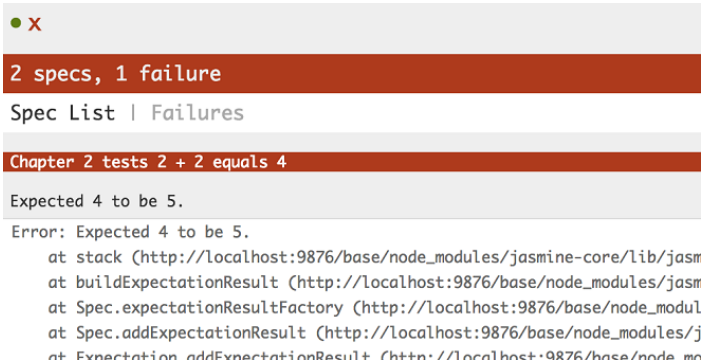

Figure 2.3 Failing tests

When you come across a failing test, it will become very that there is an error. You can use the stack trace to debug the error. Go ahead and change the `toBe` back to `4`. Now you should see a similar screent to figure 2.2.

> **NOTE** We won't using this first test going forward, so you can delete by running the following command: `rm first-jasmine-test.spec.ts`.

Now that we understand the basics behind writing tests with using Jasmine, lets move on to testing classes.

2.2 Testing classes

Writing tests for a class is the easiest and quickest way to level up our testing skills without dealing with some of the complexity that the Angular framework brings. Like we mentioned in the introduction, there are going to be real-life examples of times when you are going to test a class that is just a normal JavaScript class. Let's say that you have you a class that validates inputs from a form and you want to check the inputs client-side, before passing them along to the server. You may have a class named something like, Validator, and you can write tests just like it was any other JavaScript code without Angular at all. In this section, you are going to learn how to do that.

The first class we are going to write tests for is named `ContactClass`. The `ContactClass` is going to hold contact information for a person that we can use with the sample application. You can use the `ContactClass` to get and set a person's id, name, email, phone number, country and whether they are a favorite.

First, navigate to `website/src/app/shared/models/`. The code of the class that we will be writing test for, `ContactClass`, is in this directory in the `contact.ts` file if you want to look a peek. In the same directory, create a file called `contact.spec.ts`. Generally, it is a good idea to keep your tests in the same directory of the module you will be testing.

Typically, in test files the first thing you want to do is to import our dependencies. Since we are testing the `ContactClass` class from the `contact` module, add the code below at the top of the file:

```
import ContactClass from './contact';
```

Notice that even though the module's filename is named `contact.ts`. We left off the `ts` file extension because it is optional in `import` statements.

Next, we need to add a test suite using the `describe` method. We will call our test suite "Contact class tests". Add a blank line after the `import` statement and add in the following lines of **code**:

```
describe('Contact class tests', () => {
});
```

Inside the describe function we need to create variable to hold our instance of the `ContactClass` and set it to `null`. Add the below code inside the `describe` block.

```
let contact: ContactClass = null;
```

If you aren't used to seeing the `let` keyword instead of `var`, let me explain. The `let` keyword was introduced in ES6. `let` solves the various scoping issues that could be found using `var` because it uses block scoping instead of functional scoping. It is preferable to use `let` instead of using `var` whenever possible. To read more about `let` statements, feel free to visit the Mozilla Developer Network (MDN) at http://bit.ly/let-statment.

Now we need to initialize the `contact` variable. It is common in tests to set a variable to a reset a variable every time you execute a test. We do this because variables may have

manipulated inside an individual test. This helps to ensure that each test is run independently and previous manipulated variables do not interfere with any other tests, thus avoiding unwanted side effects.

The part of the tests that we set variables like this is known as the *Setup* part of a test. In our Setup part our test we will use the `beforeEach` method to initialize our `contact` variable every time a test is executed. Add a new line and the following code directly beneath the `contact` variable declaration that we previously added:

```
beforeEach(() => {
    contact = new ContactClass();
});
```

We will use the `beforeEach` functions to setup our tests and to execute expressions before each one of our tests are executed. In this case, we are setting the `contact` variable to a new instance of the `ContactClass` class each time a test is executed.

Now we will write our test. If an instance of `ContactClass` is created successfully it will use the class's constructor. We will test this by seeing if the `contact` is not `null`. Add a new line to directly below the `beforeEach` method that you previously added and add the below test case:

```
it('should have a valid constuctor', () => {
    expect(contact).not.toBeNull();
});
```

We can read in the message part of our `it` function, "should have a valid constructor" to see that we are trying to test the `ContactClass` constructor. We do this by evaluating the expression `expect(contact).not.toBeNull()`. Since the `Contact` class does have a valid constructor then the matcher, `not.toBeNull()`, will evaluate to true and the test will pass.

> **SHOULD YOU START TEST CASES WITH SHOULD** You may have noticed that all of our test cases thus far have started with "should". It is common syntax to start your test cases with "should do x". If you are consistent with this it makes your test cases a little easier to read. There is no harden rule on this, so you should write you test cases in a way that makes the most sense for you.

Finally, similarly to the Setup part of our tests there is a Teardown that is commonly part of tests. The Teardown part of the test can be used to make sure that instances of variable get destroyed which helps avoid memory leaks. In our case we will use the `afterEach` function to set the `contact` variable to `null`.

After the test you just added, add a new line and then add the below code:

```
afterEach(() => {
  contact = null;
});
```

Your `contact.spec.ts` file, should look like the code in listing 2.3.

Listing 2.3 contact.spec.ts – Constructor test

```
import ContactClass from './contact'; (1)

describe('Contact class tests', () => {
    let contact: ContactClass = null; (2)

    beforeEach(() => { (3)
        contact = new ContactClass();
    });

    it('should have a valid constuctor', () => { (4)
        expect(contact).not.toBeNull();
    });

    afterEach(() => { (5)
        contact = null;
    });
});
```

(1) Import ContactClass from the contact module.
(2) Declare the contact variable as a ContactClass type.
(3) Execute beforeEach function before each test case.
(4) Test the contact not to be null
(5) Execute afterEach function after each test case.

To run our new test, run the following command in your terminal window inside the `testing-angular-applications/website` directory:

```
ng test
```

In the Chrome window that is running our test runner, you should see something like the image figure 2.4:

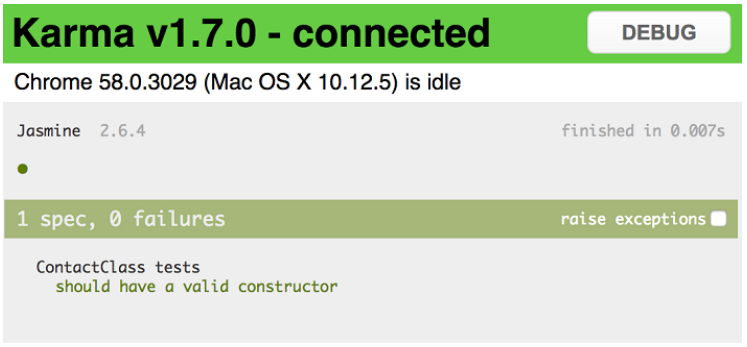

Figure 2.4: ContactClass constructor works!

Now we've tested to see that creating an empty constructor works in an instance of the `ContactClass`. Let's see what happens when we try to test creating an instance of the

`ContactClass` by passing a name to it. To see if the `name` property is set correctly we will also have to test the get method for the `name` property. Add the new test in listing 2.4 to the `contact.spec.ts` file directly below the constructor test.

Listing 2.4 contact.spec.ts – Constructor Setter and Getter name test

```
import ContactClass from './contact';

describe('ContactClass tests', () => {
    let contact: ContactClass = null;

    beforeEach(() => {
        contact = new ContactClass();
    });

    it('should have a valid constructor', () => {
        expect(contact).not.toBeNull();
    });

    it('should set name properly through constructor', () => { (1)
        contact = new ContactClass('Liz');
        expect(contact.name).toEqual('Liz');
    });

    afterEach(() => {
        contact = null;
    });
});
```

(1) Tests for setting name using a constructor and testing the getter for name.

If you left the test server running when you used `ng test`, you should find that the tests should update automatically after you add this code. If you need to restart the test server run `ng test` again. You can check to see that the tests run successfully by either looking at the test runner in the Chrome browser, as we've been doing so far, or you can check the terminal window you have been using to input commands. You should see two successfully run tests (figure 2.4).

```
05 11 2017 21:23:23.977:INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/
05 11 2017 21:23:23.977:INFO [launcher]: Launching browser Chrome with unlimited concurrency
05 11 2017 21:23:23.985:INFO [launcher]: Starting browser Chrome
05 11 2017 21:23:29.206:WARN [karma]: No captured browser, open http://localhost:9876/
05 11 2017 21:23:29.327:INFO [Chrome 62.0.3202 (Mac OS X 10.12.6)]: Connected on socket MbDP2zN5opDkacmrAAAA with id 78653198
Chrome 62.0.3202 (Mac OS X 10.12.6): Executed 1 of 2 SUCCESS (0 secs / 0.012 secChrome 62.0.3202 (Mac OS X 10.12.6): Executed
 2 of 2 SUCCESS (0 secs / 0.013 secChrome 62.0.3202 (Mac OS X 10.12.6): Executed 2 of 2 SUCCESS (0.003 secs / 0.013 secs)
```

Figure 2.5: Two successful tests

ADDING THE REST OF THE GETTERS AND SETTERS

Next, let's test out the getters and setters for the `id` and `name` private instance variables in our class. Add the two tests that are in bold in listing 2.5.

Listing 2.5 contact.spec.ts – Adding getters and setters tests for id and name tests

```
import ContactClass from './contact';

describe('ContactClass tests', () => {
       let contact: ContactClass = null;

       beforeEach(() => {
         contact = new ContactClass();
       });

       it('should have a valid constructor', () => {
         expect(contact).not.toBeNull();
       });

       it('should set name properly through constructor', () => {
         contact = new ContactClass('Liz');
         expect(contact.name).toEqual('Liz');
       });

       it('should get and set id properly', () => { (1)
         contact.id = 1;
         expect(contact.id).toEqual(1);
       });

       it('should get and set name properly', () => { (2)
         contact.name = 'Liz';
         expect(contact.name).toEqual('Liz');
       });

       afterEach(() => {
         contact = null;
       });
});
```

(1) Getters and setters for the id property.
(2) Getters and setters for name property.

If you have your test server running, you should now see four passing tests in your terminal or in your browser. If you don't have your test server running, run `ng test` in your terminal to get it started to see your passing tests.

If you want to strive for 100% test coverage for a given module, that means every line in in the module should have been covered by a test. In order for us to do this for our `ContactClass` you need to complete an exercise.

EXERCISE

For this exercise, write the rest of the tests for our `ContactClass` in the `contact.ts` file in the same directory as your test. The test should be similar to the tests that you have already written.

SOLUTION

Your final `contact.spec.ts` should look something like the code in listing 2.6.

Listing 2.6 contact.spec.ts - Complete

```
import ContactClass from './contact';

describe('ContactClass tests', () => {
       let contact: ContactClass = null;

       beforeEach(() => {
         contact = new ContactClass();
       });

       it('should have a valid constructor', () => {
         expect(contact).not.toBeNull();
       });

       it('should set name properly through constructor', () => {
         contact = new ContactClass('Liz');
         expect(contact.name).toEqual('Liz');
       });

       it('should get and set id properly', () => {
         contact.id = 1;
         expect(contact.id).toEqual(1);
       });

       it('should get and set name properly', () => {
         contact.name = 'Liz';
         expect(contact.name).toEqual('Liz');
       });

       it('should get and set email properly', () => {     (1)
         contact.email = 'liz@sample.com';
         expect(contact.email).toEqual('liz@sample.com');
       });

       it('should get and set number properly', () => {    (2)
         contact.number = '1234567890';
         expect(contact.number).toEqual('1234567890');
       });

       it('should get and set country properly', () => {   (3)
         contact.country = 'United States';
         expect(contact.country).toEqual('United States');
       });

       it('should get and set favorite properly', () => { (4)
         contact.favorite = true;
         expect(contact.favorite).toEqual(true);
       });

       afterEach(() => {
         contact = null;
       });
});
```

1. Test for getting and setting email
2. Test for getting and setting number
3. Test for getting and setting country
4. Test for getting and setting favorite

Make sure that all your tests pass; you should have eight passing tests in total. We have now reached the end of the chapter and at this point we have a pretty good understanding of how to test classes in TypeScript using the Jasmine testing framework. We covered a lot in this chapter. You should feel confident that the information learned here will help you immensely on your journey on becoming Angular testing masters.

2.3 Summary

In this chapter, you learned the following:

- **Writing basic unit tests**: Writing basic unit tests can come in handy when you do not need to use any of the Angular testing modules. Writing tests for simple functions, classes, and pipes do not require any testing dependencies outside of Jasmine, the behavior-driven development (BDD) framework. You can use Jasmine to write tests that read very close to natural English.
- **Structure of a basic test**: Most unit tests you are going to write or see in production applications will cover a similar pattern. You usually have a section at the top from importing dependencies, a section to create the test suite, a section for setting up the tests, a section for the actual tests, then a section to tear down the tests.

In the next chapter, we will start writing tests for Angular components.

3 3

Testing components

This chapter covers:

- **Testing components**
- **Knowing the differences between isolated and shallow tests**
- **Testing classes and functions**

Angular applications are built from components, so obviously the most important place to start when testing an Angular application is with component tests. For example, imagine a component that displays a calendar. It might have the ability to select a date, to change the selected date, a setting that forces the user to pick a date in the future or select the current date, and so on. Each of these use cases should be tested.

In this chapter, we'll cover key testing classes and functions such as `TestBed`, `ComponentFixture`, `async`, and `fakeAsync` that help you test your components. You'll need a good grasp of these helpers to write component tests.

Don't worry if you've never heard of these concepts before. You'll practice using them as we go; by the end of this chapter you'll understand what components are and know how to write accompanying tests for them. We'll kickoff the chapter by writing some basic component tests.

3.1 Basic component tests

The best way to get comfortable writing component tests is to write a few tests for a basic component. In this first section, we'll be writing tests for the `ContactsComponent`. The `ContactsComponent` has almost no functionality and will be easy to test.

To get started, follow the instructions for setting up the example project in Appendix A. After you complete that, from the project repo, create a file named

`contacts.component.spec.ts` in the `website/src/app/contacts/` directory, and open it in your editor. (While you're there, if you want to take a peek at the source code for the `ContactsComponent` that we'll be writing tests against, open the `contact.component.ts` file.)

The first step in creating our test is to import our dependencies. Our tests require two dependencies, the first is the `ContactsComponent` in the `contacts.component` module. At the top of the file, add the following line:

```
import { ContactsComponent } from './contacts.component';
```

The second dependency we need to import is the interface that defines a contact. Immediately after the first import statement, add the following line of code:

```
import { Contact } from './shared/models';
```

Now, we'll create the test suite that will house all our tests for `ContactsComponent`. Below the `import` statement add a `describe` block to create our test suite:

```
describe('ContactsComponent Tests', () => {
});
```

Next, we need to create a variable named `contactsComponent` that references an instance of the `ContactsComponent`. We'll set `contactsComponent` in the `beforeEach` block of our tests. Setting the `contactsComponent` variable in the `beforeEach` section will guarantee that there will be a completely new instance of the `ContactsComponent` when each test runs, preventing our test cases from interfering with each other. On the first line inside the `describe` callback function add the following code:

Let's add a `beforeEach` function that sets the `contactsComponent` variable to a new instance of the `ContactsComponent` before each test is executed:

```
beforeEach(() => {
       contactsComponent = new ContactsComponent();
});
```

Our first test just validates that we can create an instance of `ContactsComponent` properly. Add the following code below the `beforeEach` statement. After adding this code snippet, your `contacts.component.spec.ts` file should look like this the code in listing 3.1.

```
it('should set instance correctly', () -> {
       expect(contactsComponent).not.toBeNull();
});
```

This is a simple test, but if the `contactsComponent` variable contains anything other than null, the test will pass.

Listing 3.1 contacts.component.spec.ts

```
import { ContactsComponent } from './contacts.component';
import { Contact } from './shared/models';

describe('ContactsComponent Tests', () => {
  let contactsComponent: ContactsComponent = null; (1)

  beforeEach(() => {
    contactsComponent = new ContactsComponent(); (2)
  });

  it('should set instance correctly', () => {
    expect(contactsComponent).toBeTruthy(); (3)
  });
});
```

(1) Declaration of the contactsComponent variable initialized to null.
(2) A new instance of ContactsComponent will be set before each test using the beforeEach method.
(3) The assertion where you test whether the component is set correctly.

Run `ng test` to run our first test. You'll see one passing test in the Chrome window (figure 3.1). If you received an error, you should examine the error messages to see where the test is failing.

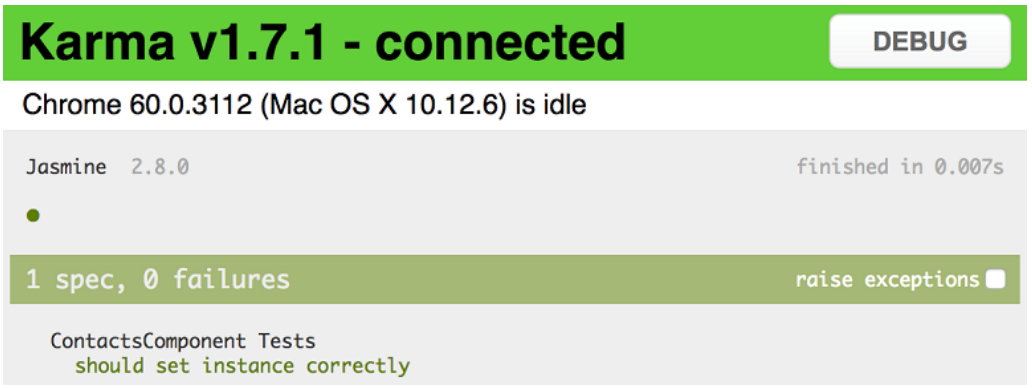

Figure 3.1: First passing unit test

Let's write a few more basic tests to finish the tests for `ContactsComponent`. For our next test, let's see what happens if there are no contacts inside the component. If there are no contacts, the `contacts` array length should be zero, because the `contacts` array is empty by default. Add the following test to our test file:

```
it('should be no contacts if there is no data', () => {
    expect(contactsComponent.contacts.length).toBe(0);
});
```

For the last test, we'll make sure that we can add contacts to the list. To do this, create a new contact using the `Contact` interface and add this contact to an array called `contactsList`. Finally, we'll set the `contacts` property of the `ContactsComponent` to the `contactsList` array that we just created. To do this, add the following code snippet below the test from before:

```
it('should be contacts if there is data', () => {
  const newContact: Contact = {
       id: 1,
       name: 'Jason Pipemaker'
  };
  const contactsList: Array<Contact> = [newContact];
  contactsComponent.contacts = contactsList;

  expect(contactsComponent.contacts.length).toBe(1);
});
```

Your completed `contacts.component.spec.ts` test should look like the code in listing 3.2.

Listing 3.2 Completed contacts.component.spec.ts file

```
import { ContactsComponent } from './contacts.component';
import { Contact } from './shared/models';

describe('ContactsComponent Tests', () => {
  let contactsComponent: ContactsComponent = null;

  beforeEach(() => {
    contactsComponent = new ContactsComponent();
  });

  it('should set instance correctly', () => {
    expect(contactsComponent).not.toBeNull();
  });

  it('should be no contacts if there is no data', () => {
    expect(contactsComponent.contacts.length).toBe(0); (1)
  });

  it('should be contacts if there is data', () => {
    const newContact: Contact = {
      id: 1,
      name: 'Jason Pipemaker'
    };
    const contactsList: Array<Contact> = [newContact];
    contactsComponent.contacts = contactsList;

    expect(contactsComponent.contacts.length).toBe(1); (2)
  });
});
```

(1) **Assertion to test that there should be no contacts by default.**
(2) **Assertion to test that if one contact is added then the number of contacts in the contact array should be one.**

If your test process is still running, you should see three passing unit tests in the Chrome test-runner window (figure 3.2).

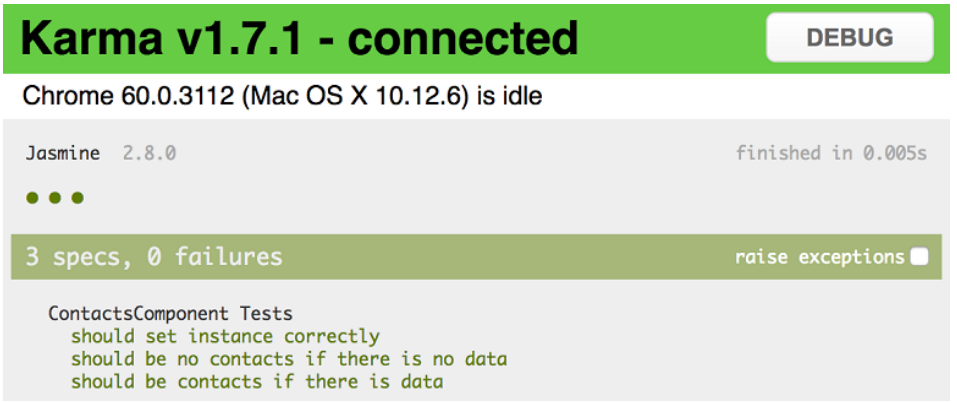

Figure 3.2 Three passing unit tests

TIP If you see any errors, check your code version in the GitHub repository at `https://github.com/testing-angular-applications/testing-angular-applications/blob/master/chapter03/contacts.component.spec.ts`.

Thus far, our tests haven't needed any Angular specific dependencies. This is because the `ContactsComponent` is just a normal TypeScript class. When testing these types of components, we don't need any help from the Angular testing modules. These types of tests are known as *isolated tests*. Isolated tests don't need any Angular dependencies and can be treated just like ordinary TypeScript files.

You might need to write tests like this when a component has very limited functionality. For example, let's say that you have just created a component for a new page, like a sign-up page, but you haven't implemented the logic yet. You could write a couple of isolated tests to ensure that the component is created correctly.

We've warmed up our component testing skills a bit, and in the next section we're going to write tests for a component with more functionality.

3.2 Real-world component testing

In the real world, you'll need to test more complex components. For example, you want to test a sidebar that contains a menu. We'd like to be able to test the sidebar without worrying about the menu. In these situations, we can use what are known as *shallow tests*. Shallow tests allow us to test components one level deep, thus ignoring any child elements that the element may contain. Using shallow tests allows you to test only the parent component in isolation.

In this section, we'll write shallow tests for the `ContactEditComponent`. The `ContactEditComponent` is similar to a component that you may come across in a real

application, so it serves as a good example to write tests against. Navigate to `website/src/app/contacts/contact-edit` in the project directory create a file named `contact-edit.component.spec.ts`. We'll begin by importing the necessary dependencies.

3.2.1 Importing the dependencies

Because the `ContactsEditComponent` is a fully functioning component, it requires a lot of dependencies. Our tests will reflect that in the number of import statements that we need. Let's consider the imports in the following order:

- testing dependencies that come from Angular
- dependencies that are included with Angular
- dependencies that we created for this project

These dependencies are required for the module to work. Let's dive into the Angular dependencies.

ANGULAR IMPORT STATEMENTS

The following is a walkthrough of the import statements that we need for our tests.

- `import { DebugElement } from '@angular/core';` – `DebugElement` can be used to inspect an element during testing. You can think of it as the native `HTMLElement` with additional methods and properties that can be useful to debug elements.
- `import { async, ComponentFixture, fakeAsync, TestBed, tick } from '@angular/core/testing';` – The `async` function takes in a function as its sole parameter. Using the `async` function you can wrap a test function in an asynchronous test zone. The test inside the `async` function will complete automatically after all asynchronous calls within the asynchronous test zone have been completed.
- `ComponentFixture` is a class that can be found in the `@angular/core` module. It can be used to create a fixture that can then be used for debugging.

  `TestBed` is a class that is used to set-up and configure our tests. Since we use `TestBed` anytime we want to write unit test for components, directives, and services it is one of the most important utilities in that Angular provides for testing. In this book, we will be using the `configureTestingModule`, `overrideModule`, and `createComponent` methods which we will put to use later in the chapter. Since the API for `TestBed` is rather extensive, we will only get to scratch the surface of the API in this book. If you want to see what else belongs to the `TestBed` API, I would recommend visiting https://angular.io/api/core/testing/TestBed.

  `fakeAsync` will run any asynchronous tasks in your test as if they were synchronous. When using `fakeAsync`, you can use `tick` to simulate the passage of time. It accepts one parameter, which is the number of milliseconds to move time forward. This allows your tests to run instantly, even if they test parts of your application that call `setTimeout`.

- `import { By } from '@angular/platform-browser';` - `By` is a class included in the `@angular/platform-browser` module that can be used to select DOM elements. For example, let's say that you want to select an element with the class name `highlight-row;` you would use the `css` method to retrieve that element using a CSS selector. The resulting code like this: `By.css('.highlight-row').`
- Note that we use a period to select the `highlight-row` class, just as you would use to set classes, ids, and elements attributes in CSS. `By` contains three methods that can be found in the table 3.1

Table 3.1: By methods

Method	Description	Parameter
all	Using all will return all of the elements.	none
css	Using a CSS attribute, you can select certain elements.	`CSS attribute`
directives	You can use a name a of directive to select directives.	Directive name

- `import { NoopAnimationsModule } from '@angular/platform-browser/animations';` - We use the `NoopAnimationsModule` class to mock animations, allowing our tests to run quickly without waiting for animations to finish.
- `import { BrowserDynamicTestingModule } from '@angular/platform-browser-dynamic/testing';` - The `BrowserDynamicTestingModule` is a module that helps bootstrap the browser to be used for testing.
- `import { RouterTestingModule } from '@angular/router/testing';` - As the name implies, the `RouterTestingModule` can be used to set up routing for testing. We include it in with our test for this component because some of the actions will involve the changing of routes.

At the top of your `contact-edit.component.spec.ts` file add the import statements from Angular (listing 3.3).

Listing 3.3 Completed contacts-edit.component.spec.ts file

```
import { DebugElement } from '@angular/core'; (1)
import { async, ComponentFixture, fakeAsync, TestBed, tick } from '@angular/core/testing';
       (2)
import { By } from '@angular/platform-browser'; (3)
import { NoopAnimationsModule } from '@angular/platform-browser/animations'; (4)
import { BrowserDynamicTestingModule } from '@angular/platform-browser-dynamic/testing'; (5)
import { RouterTestingModule } from '@angular/router/testing';    (5)
```

(1) DebugElement will be used to debug the elements we select.
(2) These dependencies are from the Angular core testing library
(3) By will be used to select elements
(4) NoopAnimationsModule should be used to simulate animations.
(5) RouterTestingModule used to test out routing.

We covered quite a bit in these `import` statements. Table 3.2 recaps the most important classes and functions that you can use for your reference when you are writing tests in this book and outside of this book.

Table 3.2: Important Testing Classes and Functions

Class / Function	Description
TestBed	`TestBed` **acts as the primary class that is used to configure unit tests when regards to Angular specific functionality. You can use** `TestBed` **to initialize and set-up your environment for unit tests.**
ComponentFixture	`ComponentFixture` **is used for testing and debugging components. It comes with many handy methods like** `destroy` **that can be used destroy a component from memory and** `isStable` **that can be used to detect when a component is stable and has completed any asynchronous tasks.**
async	**Using the** `async` **function you can wrap a test function in an asynchronous test zone. The test inside the** `async` **function will not execute until all of the asynchronous calls within the asynchronous test zone have been completed. The** `async` **function comes in useful when you are trying to run a test that contains observables or promises and you want to wait until the calls get resolved before executing the test.**
fakeAsync	`fakeAsync` **is similar to the** `async` **function in that the test will not be completed until the tasks in the asynchronous zone is executed. The difference is that the tasks will be executed in order as if they were synchronous. When using** `fakeAsync`**, you can use** `tick` **to simulate the passage of time. It accepts one parameter, which is the number of milliseconds to move time forward.**

ANGULAR NON-TESTING MODULE STATEMENT

Only one Angular non-testing module—`FormsModule`—needs to be imported. We need this module because the `ContactEditComponent` uses it for some built-in form controls. Right below the `import` statements that you just added, add in the following `import` statement:

```
import { FormsModule } from '@angular/forms';
```

REMAINING DEPENDENCY STATEMENTS

Now that we've covered the major classes and methods that we'll be using, let's add the rest of the dependencies that we need to finish the tests. Add the following lines of code below the existing imports:

```
import { Contact, ContactService, FavoriteIconDirective, InvalidPhoneNumberModalComponent,
       InvalidEmailModalComponent } from '../shared';
import { AppMaterialModule } from '../app.material.module';
import { ContactEditComponent } from './contact-edit.component';
```

```
import '../../../material-app-theme.scss';
```

Verify that your imports section looks like the code in listing 3.4 before continuing.

Listing 3.4 contacts-edit.component.spec.ts imports section

```
import { DebugElement } from '@angular/core'; (1)
import { async, ComponentFixture, fakeAsync, TestBed, tick } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { NoopAnimationsModule } from '@angular/platform-browser/animations';
import { BrowserDynamicTestingModule } from '@angular/platform-browser-dynamic/testing';
import { RouterTestingModule } from '@angular/router/testing';

import { FormsModule } from '@angular/forms'; (2)

import { Contact, ContactService, FavoriteIconDirective, InvalidEmailModalComponent,
       InvalidPhoneNumberModalComponent } from '../shared'; (3)
import { AppMaterialModule } from '../app.material.module';
import { ContactEditComponent } from './contact-edit.component';

import '../../../material-app-theme.scss'; (3)
```

(1) Angular testing dependencies
(2) Angular non-testing dependencies
(3) Dependencies created for this project

Next up, we'll set up the tests.

3.2.2 Setting up the tests

The first step in setting up our tests is to create the `describe` block that will house all our tests and declare the instance variables they need. Beneath the `import` statements add the following code snippet:

```
describe('ContactEditComponent tests', () => {
  let fixture: ComponentFixture<ContactEditComponent>;
  let component: ContactEditComponent;
  let contactService: ContactService;
  let rootElement: DebugElement;
});
```

The `describe` method creates the test suite that contains all our tests. Let's look at the instance variables. The `fixture` variable stores an instance of the `ComponentFixture`, which contains methods that help us debug and test a component. The `component` variable stores an instance of the `ContactEditComponent`. The `contactService` is a fake instance of `ContactService` (more on that shortly). The `rootElement` stores the `DebugElement` for our component, which is how we'll access its children.

INFO A test fake is an object used in a test that substitutes for the real thing. A mock is a fake that simulates the real object and keeps track of when it's called and what arguments it receives. A stub is a simple fake with no logic, that always returns the same value.

We use a test fake for the `ContactService`, because the real `ContactService` makes HTTP calls, which would make our tests harder to run and less deterministic. Also, faking the `ContactService` allows us to focus on testing the `ContactEditComponent`, without worrying about how `ContactService` works. Angular's dependency injection system makes it easy to instantiate a `ContactEditComponent` with a fake version of `ContactService`. The fake `ContactService` has the same type as the real one, so the TypeScript compiler will throw an error if we forget to stub out part of the interface.

Right below the last variable declaration, but still inside the `describe` block, add the code in in listing 3.5 to create our fake service named `contactServiceStub`:

Listing 3.5 Mock ContactService

```
const contactServiceStub = {
  contact: { (1)
    id: 1,
    name: 'janet'
  },

  save: async function (contact: Contact) { (2)
    component.contact = contact;
  },

  getContact: async function () { (3)
    component.contact = this.contact;
    return this.contact;
  },

  updateContact: async function (contact: Contact) { (4)
    component.contact = contact;
  }
};
```

(1) The default contact object.
(2) Set's the passed in object to the component's contact property.
(3) Method that sets the current contact to the component's contact property and returns that contact.
(4) Method that updated the contact object.

Now that we have a fake `ContactService`, let's add two `beforeEach` blocks, which be executed before each test. The first `beforeEach` sets up our `TestBed` configuration. The second will set our instance variables. We could have just one `beforeEach`, but our test is easier to read if we keep them separate.

There is a lot going on here, so let's break down the code in listing 3.6 a bit. The `TestBed` class has a method called `configureTestingModule`. You can probably guess it's purpose, which is to configure the testing module. The `configureTestingModule` is much like the

`NgModule` class that is include in the `app.module.ts` file, which can be found at `src/app`. The only difference is that `configureTestingModule` is used only in tests. The `configureTestingModule` takes an object that is in the format of a `TestModuleMetadata` *type alias*. If you aren't familiar with a type alias, think of it like an interface for our purposes. In the code listing, note also the providers section:

```
providers: [{provide: ContactService, useValue: contactServiceStub}]
```

This is where we provide our fake contact service `contactServiceStub` in place of the real `ContactService` with `useValue`.

We use `overrideModule` in this case because we need the two modal dialogs to be loaded lazily. Currently, the only way to do this is to use `overrideModule` and set the `entryComponents` value to an array that contains the two modal components that the `ContactEditComponent` uses - `InvalidEmailModalComponent` and `InvalidPhoneNumberModalComponent`.

Finally, the last line of our first `beforeEach` statement uses `TestBed.get(ContactService)` to get a reference to our fake `contactService` from Angular's dependency injector. This will be the same instance that the `ContactEditComponent` uses.

After the code for the `contactServiceStub`, add the code in Listing 3.6 as our first `beforeEach` statement.

Listing 3.6 First beforeEach

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ContactEditComponent, FavoriteIconDirective, InvalidEmailModalComponent,
       InvalidPhoneNumberModalComponent],
    imports: [
      AppMaterialModule,
      FormsModule,
      NoopAnimationsModule,
      RouterTestingModule
    ],
    providers: [{provide: ContactService, useValue: contactServiceStub}] (1)
  }); (2)

  TestBed.overrideModule(BrowserDynamicTestingModule, {
    set: {
      entryComponents: [InvalidEmailModalComponent, InvalidPhoneNumberModalComponent]
    }
  }); (3)

  contactService = TestBed.get(ContactService); (4)
});
```

(1) **This is where we use the contactServiceStub instead of the real service.**
(2) **Configuring TestBed to be used in our tests.**
(3) **We have to use overrideModule because there are a couple of components that will be lazy loaded.**

❹ Getting an instance of the contactService that uses the contactServiceStub.

In our example, you can see that `TestModuleMetadata` accepts four optional properties, which are described in Table 3.3.

Table 3.3: TestModuleMetadata optional fields

Field	Data Type	Description
declarations	any[ ]	This is where you list any components that the component that you are testing may need.
imports	any[ ]	`imports` should be set to an array of modules that the component that you are testing requires.
providers	any[ ]	Lets you override the providers Angular uses for dependency injection. In our case, we inject a fake ContactService.
schemas	Array<SchemaMetadata | any[ ]>	You can use schemas like `CUSTOM_ELEMENTS_SCHEMA` and `NO_ERRORS_SCHEMA` to allow for certain properties on elements. For example, the `NO_ERRORS_SCHEMA` will allow for any property to be on any element that is going to get tested. **NOTE** Beware, using the `NO_ERRORS_SCHEMA` can cover up issues, because Angular won't complain if your template has invalid elements or properties.

Now let's add the second `beforeEach` statement. The `fixture` variable stores the component-like object from the `TestBed.createComponent` method that we can use for debugging and testing that we mentioned earlier. The `component` variable holds a component that we get from our `fixture` using the `componentInstance` property.

But what is this `fixture.detectChanges` method that we haven't seen before? The `detectChanges` method triggers a change-detection cycle for the component; you need to call it after initializing a component and or changing a data-bound property value. After calling `detectChanges`, the updates to your component will be rendered in the DOM. In production, Angular uses something called Zones (which we'll learn more about in Chapter 9) to know when to run change detection, but in unit tests we don't have this mechanism. Instead, we need to call `detectChanges` frequently in our tests after we make changes to a component.

Directly below the first `beforeEach` statement add in the following code snippet:

```
beforeEach(() => {
  fixture = TestBed.createComponent(ContactEditComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
```

```
  rootElement = fixture.debugElement;
});
```

So far, so good. We've added the code to set up our tests. In the next section, we'll add the actual tests.

3.2.3 Adding the tests

We're ready to write our tests. We want to test the `saveContact`, `loadContact`, and `updateContact` methods for `ContactEditComponent` since those methods contain most of the functionality of the component. There are several more helper private methods inside the `ContactEditComponent` class, but we don't care about those because testing the component's public API will exercise them. In general, you shouldn't test private methods; if a method is important enough to be tested, you should consider making it public.

First, let's write a test for the `saveContact` method. Calling `saveContact` change the component's state, which will be reflected in changes to the DOM. We'll use the `fakeAsync` method, so that the test won't finish until the component has finished updating. If we try to test our code `fakeAsync` or `async`, our test might finish before asynchronous calls have a chance to complete. If we use `fakeAsync`, our test will fail if there are still asynchronous task pending when the test is done.

Next, we create a contact object and set the `component.isLoading` property to `false`. We need to do this manually, otherwise all that will render is the loading-progress bar. Then we'll call the `saveContact` method to save the contact that is stored in the `contact` variable. Normally, `saveContact` would use the real `ContactService`, but because we configured the testing module to provide `contactServiceStub` earlier, the component will call the stub.

After the `saveContact` method is called, you'll notice that we call `detectChanges`. As we mentioned earlier, after you make changes to components, you need to call `detectChanges` so that those changes will be rendered, which allows us to test that changes to the component are reflected in the DOM.

After calling `detectChanges` we query `rootElement` using `By.css` for the `contact-name` class to get the input element that contains the contact name. The we call `tick` to simulate the passage of time, so that the component will finish updating. Finally, we assert that the value of `nameInput` is equal to '`lorace`'.

Copy the code in listing 3.7 just below the last `beforeEach` statement. Make sure that you stay within the overall test suite (the top-level `describe` block).

Listing 3.7 saveContact method test

```
describe('saveContact() test', () => {
  it('should display contact name after contact set', fakeAsync(() => {
    const contact = { (1)
      id: 1,
      name: 'lorace'
    };

    component.isLoading = false; (2)
```

```
    component.saveContact(contact); ❸
    fixture.detectChanges(); ❹
    const nameInput = rootElement.query(By.css('.contact-name')); ❺
    tick(); ❻
    expect(nameInput.nativeElement.value).toBe('lorace'); ❼
  }));
});
```

❶ The contact object that we will save.
❷ Set the isLoading to false to hide the progress bar.
❸ Save the contact object.
❹ Use the detectChanges method to trigger change detection.
❺ Get the nameInput form field.
❻ Simulate the passage of time using tick.
❼ The assertion checks to see if the name property has been set correctly.

Next, we'll write a test for the `saveContact` method. This test is very similar to the test in Listing 3.5. The only difference is that we use the `loadContact` method instead of the `saveContact` method of the `ContactEditComponent` class. The `loadContact` method will load a contact for our testing purposes inside the `contactServiceStub`. The contact's name is "`janet`" and which is the value we use in the assertion.

Copy the code in Listing 3.8 directly below the `saveContact` method test that we just created.

Listing 3.8 loadContact method test

```
describe('loadContact() test', () => {
  it('should load contact', fakeAsync(() => {
    component.isLoading = false;
    component.loadContact(); ❶
    fixture.detectChanges();
    const nameInput = rootElement.query(By.css('.contact-name'));
    tick();
    expect(nameInput.nativeElement.value).toBe('janet'); ❷
  }));
});
```

❶ Executes the loadContact method.
❷ The default contact that is loaded has a value of "janet" for the name property. We expect the name field to have the contact's name which is "janet".

Now we'll move on to testing the `updateContact` method.

3.2.4 Adding the updateContact tests

By now you've probably picked up on a pattern: this test is similar to the other two tests. This time we initially set a contact that has a name of "`chauncey`" and test that the component renders correctly. The major difference between this test and the other two tests is that there

is a second assertion. We want to check to see that the name updates when we call `updateContact`. To do this, we call `updateContact` and pass it `newContact`.

You might notice that we call `tick` in the following listing with 100 as a parameter. This is because the `updateContact` method takes a bit longer to execute than the other methods that we've been testing. Copy the code in listing 3.9 below the previous test.

Listing 3.9 First updateContact method test

```
describe('updateContact() tests', () => {
  it('should update the contact', fakeAsync(() => {
    const newContact = {
      id: 1,
      name: 'london',
      email: 'london@example.com',
      number: '1234567890'
    };

    component.contact = {
      id: 2,
      name: 'chauncey',
      email: 'chauncey@example.com',
      number: '1234567890'
    };

    component.isLoading = false;
    fixture.detectChanges();
    const nameInput = rootElement.query(By.css('.contact-name'));
    tick();
    expect(nameInput.nativeElement.value).toBe('chauncey');

    component.updateContact(newContact);                    (1)
    fixture.detectChanges();                                (2)
    tick(100);                                              (3)
    expect(nameInput.nativeElement.value).toBe('london');   (4)
  }));
});
```

(1) Updates the existing contact to the newContact object.
(2) We use the detectChanges method to trigger change detection.
(3) The tick method simulates the passage of time. In this case we simulate the passage of 100 milliseconds.
(4) In the assertion we are checking to see that the value in name input form field has been changed correctly.

Run `ng t` in your console (if you haven't already); you should see six passing tests. If you don't have six passing tests, go back to the code samples and make sure that your code matches the code that in the book.

We now have a test that will update a contact, but we need to test what happens when we try to update the contact with invalid contact data. First, let's see what happens when we try to update the contact with an invalid email address. The differences between listing 3.10 and 3.9 are highlighted in bold. The `newContact` variable now has an invalid email, and the last assertion does not expect the contact to change because the email is invalid. That is why both

assertion expect the contact's name to remain "`chauncey`". Add the code in listing 3.10 directly below the first `updateContact` method test.

Listing 3.10 Second updateContact method test

```
it('should not update the contact if email is invalid', fakeAsync(() => {
  const newContact = {
    id: 1,
    name: 'london',
    email: 'london@example', (1)
    number: '1234567890'
  };

  component.contact = {
    id: 2,
    name: 'chauncey',
    email: 'chauncey@example.com',
    number: '1234567890'
  };

  component.isLoading = false;
  fixture.detectChanges();
  let nameInput = rootElement.query(By.css('.contact-name'));
  tick();
  expect(nameInput.nativeElement.value).toBe('chauncey');

  component.updateContact(newContact);
  fixture.detectChanges();
  tick(100);
  expect(nameInput.nativeElement.value).toBe('chauncey'); (2)
}));
```

(1) Email is invalid.
(2) Since the email is invalid the contact should not be updated using the newContact object, therefore the contact name should be the same.

Now let's see what happens when we try to update a contact with an invalid phone number. Again, notice the bolded code in the listing that follows. The only difference between this test and the previous test is that the number now contains too many digits. Similar to the test before this one, the contact name is the same in both assertions.

Copy the code in listing 3.11 to the end of the second `updateContact` method test that you just wrote.

Listing 3.11 Third updateContact method test

```
it('should not update the contact if phone number is invalid', fakeAsync(() => {
  const newContact = {
    id: 1,
    name: 'london',
    email: 'london@example.com',
    number: '12345678901' (1)
  };
```

```
  component.contact = {
    id: 2,
    name: 'chauncey',
    email: 'chauncey@example.com',
    number: '1234567890'
  };

  component.isLoading = false;
  fixture.detectChanges();
  let nameInput = rootElement.query(By.css('.contact-name'));
  tick();
  expect(nameInput.nativeElement.value).toBe('chauncey');

  component.updateContact(newContact);
  fixture.detectChanges();
  tick(100);
  expect(nameInput.nativeElement.value).toBe('chauncey'); ❷
}));
```

❶ **Number is invalid.**
❷ **Since the number is invalid the contact should not be updated using the newContact object, therefore the contact name should be the same.**

Run `ng t` in your terminal; you should see eight passing tests. If you see any errors, try checking you code version the code in the GitHub repository at: `https://github.com/testing-angular-applications/testing-angular-applications/blob/master/chapter03/contacts-edit.component.spec.ts`.

We've coded complete test coverage for a real-world component! In the future, you may come across components that are more advanced, but what you learned forms the foundation for writing those tests. Components are one of the most—if not the most—important concepts in Angular. With that said, you should have a firm understanding of the basics to be successful in writing tests.

3.3 Summary

In this chapter, you learned the following:

- *Shallow tests* versus *isolated tests*. Isolated tests don't rely on the built-in Angular classes and methods. They can be tested as if there are normal JavaScript classes. Sometimes our tests will require the actual rendering of components one level deep without requiring the rendering of child components. These types of tests are known as shallow tests.
- Using the `async` function, you can wrap a test function in an asynchronous test zone. The test inside the `async` function will complete automatically after all asynchronous calls within the asynchronous test zone have been completed.
- Use the `ComponentFixture` class to debug an element.
- `TestBed` is a class that is used to set-up and configure our tests. We use it anytime we want to write unit test that test out components, directives, and services.

- `DebugElement` can be used to dive deeper into an element. You can think of it as the `nativeElement` with additional methods and properties that can be useful to debug elements.
- The `nativeElement` object is an Angular wrapper around the built-in DOM native element.

Now that you understand how to test our components, in the next chapter we'll look at how to test Angular Directives.

4

Testing directives

This chapter covers:

- **Using the types of directives available in Angular**
- **Testing attribute and structural directives**
- **Using `TestMetaData` to configure `TestBed`**

In this chapter, you will learn how to test directives. *Directives*, like components, are a way to encapsulate parts of your application into reusable chunks of code. Both directives and components allow you to add behavior to the HTML in your application; the only difference between directives and components is that components contain a view.

For example, let's say your application has a table, and that you want to change the background color of a row when the pointer hovers over it. You could create a directive, perhaps named `HighlightRowDirective`, that adds that row highlighting behavior and reuse it throughout your project.

Before we get started writing the tests for directives, let's learn some more about them.

4.1 What are directives?

In Angular, there are three types of directives:

- Components
- Structural directives
- Attribute directives

We already covered components in the previous chapter, but to review, components are chunks of code that can be used as HTML tags and can be placed throughout an application.

4.1.1 Components versus directives

Components are a type of directive, and as mentioned before, the only difference between directives and components is that a component contains a view (defined by a template). Another way to think about the difference is that components are visible, while a directive modifies the element it's attached to.

To further expand on our understanding, let's look at two of the decorators that are included with the Contacts application that we have been working on. *Decorators* are a way to add behavior to a class or method, kind of like annotations in Java.

First, let's look at the `@Component` decorator for the `ContactDetail` component that can be found at `/website/src/app/contacts/contact-detail/contact-detail.component.ts`:

```
@Component({
  selector: 'app-contact-detail',
  templateUrl: './contact-detail.component.html',
  styleUrls: ['./contact-detail.component.css']
})
```

There are a few options to customize the view, such as `templateUrl`, `styleUrls`, and `viewProviders`. Now, let's look at the `@Directive` decorator for `FavoriteIcon`, which can be found in `/website/src/app/contacts/shared/favorite-icon/favorite-icon.directive.ts`, which we'll be testing later in this chapter.

```
@Directive({
  selector: '[appFavoriteIcon]'
})
```

The `FavoriteIcon` `@Directive` decorator has a selector, just like the `@Component` decorator, but there are no options for `templateURL`, `styleUrls`, or `viewProviders`. This is because there are no views associated with directives, and thus there are no templates to use, create, or style.

Now that we've looked at components and the difference between components and directives, let's review the difference between structural and attribute directives.

4.1.2 Attribute directives

Attribute directives are used when you are trying to change the appearance of a DOM element. The example of the directive that we talked about in the introduction is a good example of an attribute directive where we were changing the background color of a row in a table gray and highlight it as a user rolls over a row that we mentioned earlier.

4.1.3 Structural directives

Structure directives are used to add or remove elements from the DOM – in other words, to change the structure of the page. Angular includes a few structural directives out of the box, like `ngIf` and `ngShow`.

4.1.4 Our directives

In this chapter, we will create tests for both an attribute directive and a structural directive. The attribute directive adds a gold star to a contact when it's marked as a favorite. The structural directive will add and remove contact tables depending on whether there are contacts available.

First, we'll write tests for the attribute directive, and then we'll move on to the structural directive.

4.2 Testing attribute directives

To test an attribute directive, we simply get an instance of the directive, take some kind of action, and then check that the expected changes show up in the DOM. Before we get to the actual process of writing the tests, let's take a closer look at the attribute directive that we will be testing.

4.2.1 Introducing the Favorite Icon directive

We'll be testing a directive that I created, named `FavoriteIconDirective`. The source code for the `FavoriteIconDirective` is in `/website/src/app/contacts/shared/favorite-icon/icon.directive.ts`. The `FavoriteIconDirective` can be added to an element to display a star when a contact is favorited. Let's see how its used.

BASIC USAGE

Usage of the Favorite Icon looks like the following:

```
<element [appFavoriteIcon]="expression"></element>
```

Here is a working example of using the `FavoriteIconDirective` taken from `contact-list.component.html` template on line 20 at `/website/src/app/contacts/contacts-list/` in the Contacts application source code:

```
<i [appFavoriteIcon]="contact.favorite"></i>
```

In the above example, you can see that simply set `[appFavoriteIcon]` to the `contact.favorite` expression which can either be `true` or `false`. If the expression evaluates to `true`, meaning that the contact is a favorite, there will be a gold star displayed as in figure 4.1.

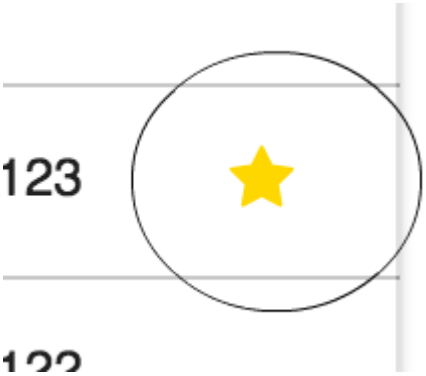

Figure 4.1 Favorite Icon set to true

WHAT'S THE DEAL WITH THOSE BRACKETS? Glad you asked! In this case, we use the brackets to bind an expression to our directive. There are a couple of different ways you can use brackets for binding in Angular. Check out the Binding Syntax in the Angular docs at https://angular.io/docs/ts/latest/guide/template-syntax.html#!#binding-syntax for more information.

Figure 4.2 shows what happens when the bound expression is false. The star becomes white, which makes it invisible against the white background.

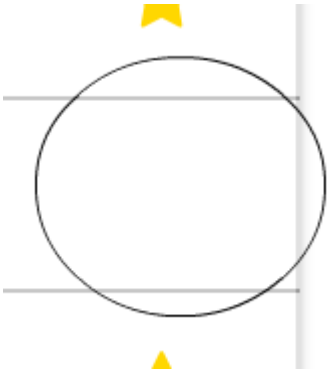

Figure 4.2: Favorite Icon set to false with a white star displayed.

If you roll over a row where the Favorite Icon is set to false, you will be able to see the white star because the background color changes to gray, as in figure 4.3.

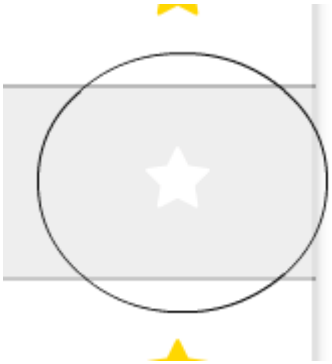

Figure 4.3: Favorite Icon set to false with white star displayed while row is hovered over.

ADDING COLOR AS A SECOND PARAMETER

The `FavoriteIconDirective` defaults to the color gold, but you can pass in a second parameter that changes the color of the star.

Setting the color parameter of Favorite Icon looks like this:

```
<element [appFavoriteIcon]="expression" [color]="'color name'"></element>
```

In figure 4.4 we display a blue star when the Favorite Icon expression evaluates to `true` and the color parameter is set to using the following code:

```
<i [appFavoriteIcon]="contact.favorite" [color]="'blue'></i>
```

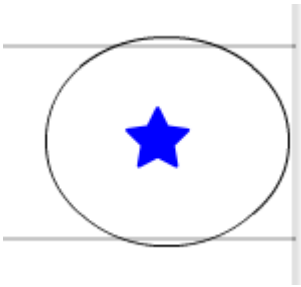

Figure 4.4: Favorite Icon set to true with solid, blue star displayed when the color parameter is set.

The rest of the cases are the same as the default star. Now that we understand the functionality of the Favorite Icon directive, let's go over the test cases and then we can start writing tests!

4.2.2 Creating tests for the favorite icon directive

Let's separate our test cases into three different parts:

- When the Favorite Icon directive is set to `true`
- When the Favorite Icon directive is set to `false`
- When a color parameter is passed in as a second parameter

The top two sets of cases will be the most common usage of the directive and then we'll have a third set of tests for when the color of the star is changed to blue.

WHEN THE FAVORITE ICON DIRECTIVE IS SET TO TRUE

In Table 4.1, we start looking at test cases when the Favorite Icon directive is set to `true`.

Table 4.1: When the Favorite Icon directive is set to true test cases.

Test Case	Event	Displays
The element should include a gold star after the page loads.	After page loads	Gold star
The element should still display a gold star if the user rolls over the star.	Roll over	Gold star
The element should still display the black outline of a star after the user clicks on the star.	On click	Black outline star
The element should still display a gold star if the user rolls off the star.	Roll off	Gold star

WHEN THE FAVORITE ICON DIRECTIVE IS SET TO FALSE

In Table 4.2, we start looking at test cases when the Favorite Icon directive is set to `false`.

Table 4.2: When the Favorite Icon directive is set to false test cases.

Test Case	Event	Displays
The element should include a white star after the page loads.	After page loads	White star
The element should display the black outline of a star if the user rolls over the star.	Roll over	Outline star
The element should contain a gold star after the user clicks on the star.	On click	Gold star
The element should contain a white star after the user rolls off the star.	Roll off	White star

WHEN THE COLOR PARAMETER IS SET TO A COLOR

Table 4.3 includes the test cases that we will need to check for to ensure that the color parameter works as expected.

Table 4.3: When the color parameter is set to a color test cases.

Test Case	Event	Displays
The element should include the color that is specified in the second parameter star after the page loads.	After page loads	Specified color star
If there is an unrecognized color, thee color of the star should be set to black by default.	After page loads	Black star

4.2.3 Setting up the favorite icon directive test suite

Now that we've planned out the test cases, let's create our test suite. Create a file named `favorite-icon.directive.spec.ts` in the `/website/src/app/contacts/shared/favorite-icon` directory. First, we import the dependencies that we will be using to execute our tests. Add the following statements at the top of your file to import the Angular dependencies:

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed, TestModuleMetadata } from '@angular/core/testing';
```

Now that we've imported the Angular dependencies, let's import the dependencies that were created for the contacts application:

```
import { constants } from './favorite-icon.constants';
       import { FavoriteIconDirective } from './favorite-icon.directive';
import { getStarElement, doClassesMatch } from '../../testing';
```

To test a directive, we need to create a host component that uses it. Our host component has a different `<i>` element for each test case: Add the below code right below the import statements:

```
@Component({
  template: `
      <i [appFavoriteIcon]="true"></i>
      <i [appFavoriteIcon]="false"></i>
      <i [appFavoriteIcon]="true" [color]="'blue'"></i>
      <i [appFavoriteIcon]="true" [color]="'cat'"></i>
    `
})
class TestComponent { }
```

Now create a `describe` block that will house all our tests for the `FavoriteIconDirective`. Add the below line of code right below the `TestComponent` class that we just created.

```
describe('Directive: FavoriteIconDirective', () => {
});
```

Inside the `describe` block, let's create some variables that we'll use in all the Favorite Icon tests:

```
let fixture: ComponentFixture<any>;
const expectedSolidStarList = constants.classes.SOLID_STAR_STYLE_LIST;
const expectedOutlineStarList = constants.classes.OUTLINE_STAR_STYLE_LIST;
```

We'll set the fixture variable in the beforeEach block, so that there's a fresh fixture created for each test. Add the `beforeEach` the variables we just declared (listing 4.1).

Listing 4.1 First Jasmine test

```
beforeEach(() => {
  const testModuleMetadata: TestModuleMetadata = {                    (1)
    declarations: [FavoriteIconDirective, TestComponent]
  };
  fixture = TestBed.configureTestingModule(testModuleMetadata)         (2)
                   .createComponent(TestComponent);                    (3)
  fixture.detectChanges();                                             (4)
});
```

(1) We declare the testModuleMetadata to contain the information needed to configure TestBed.
(2) We configure TestBed using the testModuleMetadata variable.
(3) We use TestBed.createComponent to create a component fixture to use with our tests.
(4) We use detectChanges to initiate change detection.

Let's take a second to recap what is included in the `beforeEach` statement. If you read chapter 3, then some of this may look familiar.

In the first line of the `beforeEach` method we declare a variable called `testModuleMetadata`. `testModuleMetadata` implements the `TestModuleMetadata` interface which we use to provide test metadata to configure the `TestBed`. In the previous chapter, we used test metadata to configure `TestBed`. The difference this time is that we created a separate variable to contain that data. In that case instead, we passed an object that conformed to the `TestModuleMetadata` interface to the `configureTestingModule` method that configured `TestBed`.

After we configure `TestBed`, we use `createCompoment` method from `TestBed` to return an instance of a `ComponentFixture`. Finally, we call `fixture.detectChanges()` to invoke change detection and render updated data whenever an event occurs, such as `click` or `mouseenter`.

Now let's add an `afterEach` block after the `beforeEach` block to ensure that the fixture object is destroyed by setting it to `null`:

```
afterEach(() => { fixture = null; });
```

We have now completed setting up the tests. We can move on to the writing the actual tests.

4.2.4 Creating the Favorite Icon directive tests

Almost all our tests will follow a similar pattern. We will create a new instance of a component, run some kind of event, and then finally check that the element changed as expected.

TESTS FOR WHEN THE FAVORITE ICON DIRECTIVE IS SET TO TRUE

First off, let's create a `describe` block directly after the `afterEach` code we just added, so that we can group together all the tests that cover when the Favorite Icon is set to true.

```
describe('when favorite icon is set to true', () => {
});
```

We create a variable named `starElement` to reference the star element and set it to `null`. We initialize the variable to `null` because we'll set it later in the `beforeEach` block that will be executed before each test. Add this line right below the `describe` block:

```
let starElement = null;
```

Now we will create another `beforeEach` method that is scoped only to this suite. Put the following code below the variable we just declared:

```
beforeEach(() => {
  const defaultTrueElementIndex = 0;
  starElement = getStarElement(fixture, defaultTrueElementIndex);
});
```

Notice that in the line after `beforeEach` we declare a variable named `defaultTrueElementIndex` and set it to `0`. You may recall that earlier when we created the `TestComponent` that the template contained four different sets of HTML tags for the different test cases. The different elements are stored in an array. We are testing the first element in the `fixture` for this set of tests, so we use the `0` index to retrieve it from the array. Recall, that `fixture` was created by using the `TestComponent` class.

To get the `starElement` from the fixture we use a helper method called `getStarElement`. All the `getStarElement` method does is extract a child element from the fixture. If you are curious about the implementation, you can read the source code at `/website/src/app/contacts/testing/get-star-element.ts`.

Finally, let's create the `afterEach` method that will set the `starElement` to `null`:

```
afterEach(() => { starElement = null; });
```

Let's first check to see if the favorite icon appears after the page loads.

THE ELEMENT SHOULD INCLUDE A SOLID GOLD STAR AFTER THE PAGE LOADS

To start off, we'll create an `it` block and place it after the `beforeEach` block we just added:

```
it('should display a solid gold star after the page loads', () => {
});
```

Our first test case will check that the element's color is gold, as expected. Inside the `it` block add in the below code:

```
expect(starElement.style.color).toBe('gold');
```

For our second test, we'll check that the colors list matches the colors in our elements list. To do this, we have another helper method called `doClassesMatch`. Just as with the `getStarElement` method, this method can be found at `/website/src/app/contacts/testing/`. All this method does is take an element and a list of styles, and makes sure they match by looping through the styles in the lists. The result of the comparison is true if the element has all the expected styles.

The style classes for a solid star are stored as a list called `expectedSolidStarList`. If you were to look at the contents of this list you would find three classes: `['fa', 'fa-star', 'fa-lg']`. These are all the classes that can be expected for a solid star. We would expect our `starElement` to include these classes to be correctly styled.

That being said, let's add the following code to our test:

```
expect(doClassesMatch(starElement.classList, expectedSolidStarList)).toBeTruthy();
```

Your completed test should look like this:

```
it('should display a solid gold star after the page loads', () => {
  expect(starElement.style.color).toBe('gold');
  expect(doClassesMatch(starElement.classList, expectedSolidStarList)).toBeTruthy();
    });
```

Now in your terminal, run the following command:

```
ng test
```

You should see something similar to figure 4.5.

```
21 10 2016 09:09:06.123:INFO [karma]: Karma v1.2.0 server started at http://localhost:9876/
21 10 2016 09:09:06.124:INFO [launcher]: Launching browser Chrome with unlimited concurrency
21 10 2016 09:09:06.129:INFO [launcher]: Starting browser Chrome
21 10 2016 09:09:06.898:INFO [Chrome 53.0.2785 (Mac OS X 10.11.6)]: Connected on socket /#L_sISeVv8TCyMBUdAAAA with id 78894617
Chrome 53.0.2785 (Mac OS X 10.11.6): Executed 1 of 1 SUCCESS (0.132 secs / 0.109 secs)
```

Figure 4.5: First test successfully executed

The element should still display a solid gold star if the user rolls over the star

Our second test is a bit more complicated than the first test, because we need to simulate a rollover effect. We can use the `Event` class to create a `mouseenter` event to simulate the user moving the pointer over the star. We manually dispatch the event by using the `dispatchEvent` method that's part of every DOM element. Add the below code underneath our first test:

```
it('should display a solid gold star if the user rolls over the star', () => {
       const event = new Event('mouseenter');
       starElement.dispatchEvent(event);
});
```

Our two test cases are the same as the first test case, because we still expect the gold star to still be shown when the user hovers over it. Add the following code after the event code that we just added:

```
expect(starElement.style.color).toBe('gold');
expect(doClassesMatch(starElement.classList, expectedSolidStarList)).toBeTruthy();
```

Now run `ng test`. You should now have two completed tests.

THE ELEMENT SHOULD STILL DISPLAY THE OUTLINE OF A BLACK STAR IF THE USER CLICKS ON THE STAR

This test is similar to the previous test. The only difference is that because this is a `click` event, we use a different argument when we create an instance of the `Event` class. We also change the expected color to be `black` and the class list to `expectedOutlineStarList`, because we expect the star to be only as an outline instead of a solid star.

Because there are only small changes, we'll present the full test below with the changes in bold and you can add this test after the previous test:

```
it('should display a black outline of a star after the user clicks on the star', () => {
       const event = new Event('click');
       starElement.dispatchEvent(event);

       expect(starElement.style.color).toBe('black');
       expect(doClassesMatch(starElement.classList, expectedOutlineStarList)).toBeTruthy();
});
```

Execute the tests again in your terminal by running `ng test` and you should now have three successful tests. We have covered all the tests for our attribute directive. The completed test is at `/chapter04/favorite-icon.directive.spec.ts` for your reference. Next up, we will take a look at how to test structural directives.

4.3 Testing structural directives

Testing structural directives is similar to testing attribute directives. We're just checking that the DOM is rendered as we expect when the directive is used. Before we start writing tests, let's look at the directive we will be testing, the `ShowContactsDirective`.

4.3.1 Introducing the ShowContactsDirective

The structural directive we'll be testing is `ShowContactsDirective`. The `ShowContactsDirective` can be used to add or remove elements to the DOM. It mimics the implementation of `ngIf`; we're using it only to demonstrate how to test structural directives.

USAGE

Here is an example of using the `ShowContactsDirective`:

```
<div *appShowContacts="contacts.length"></div>
```

In the above example, you can see that we simply set `*appFavoriteIcon` to the `contacts.length` expression, which in JavaScript resolves to `true` if the length is greater than `1` and `false` if the length is `0`. This example is taken directly from the application and the code can be found at `/website/src/app/contacts/contact-list.component.html` on the first line if you would like to see it for yourself.

> **WHAT'S THE DEAL WITH THE ASTERISK?** The asterisk transforms the element the directive is attached to into a template. The directive then controls how that template is rendered to the DOM, which is how it can alter the structure of the page. To learn more about the asterisk prefix, check out https://angular.io/guide/structural-directives#asterisk.

To see how the directive is used, start the application using `ng s` and then open your browser to http://localhost:4200/ and you should see something like the screen in figure 4.6:

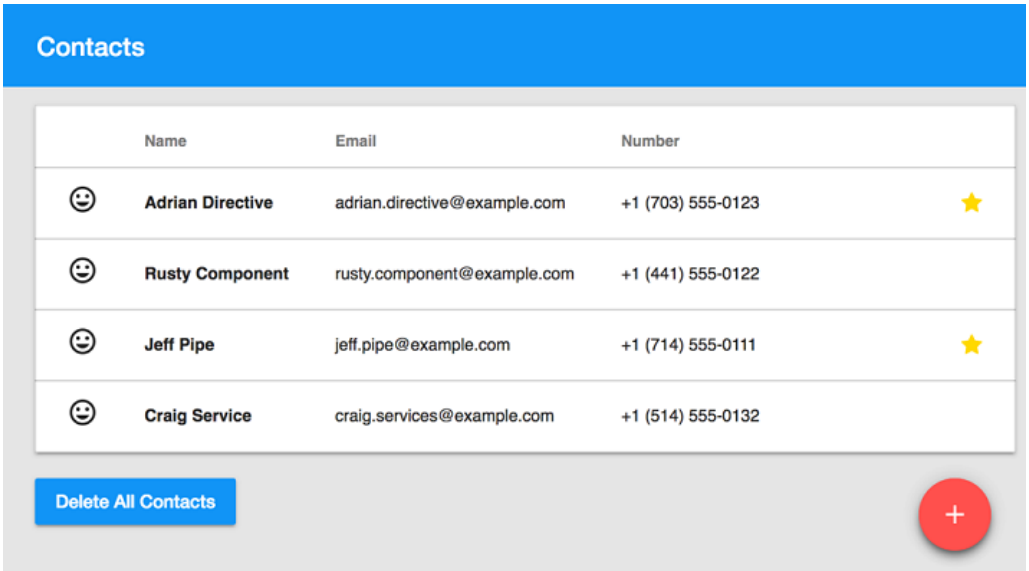

Figure 4.6: The Contacts application with contacts

Now click the "Delete All Contacts" button. You should see the screen in figure 4.7:

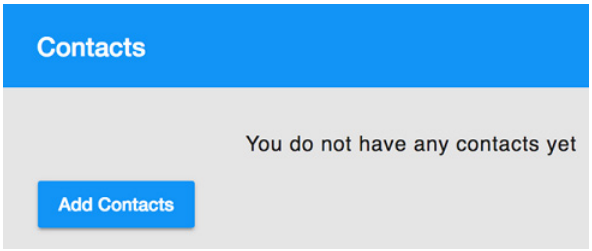

Figure 4.7: The Contacts application with no contacts

Notice that the table completely disappeared and now there is a message stating, "You do not have any contacts yet." That is because the `ShowContactsDirective` is hiding and showing elements based on whether there are any contacts. The "Delete All Contacts" text has changed as well to "Add Contacts." There are two different buttons are shown or hidden using the `ShowContactsDirective`. Go ahead and click "Add Contacts" and you'll see the same screen that we saw in figure 4.8.

If you would like to look at the `ShowContactsDirective` source code, navigate to `/website/src/app/contacts/shared/show-contacts/` and view the `show-contacts.directive.ts` file. Now that we understand the basic functionality of the `ShowContactsDirective`, let's go over all the test cases that we will be writing tests for.

4.3.2 Creating our tests for the Show Contacts directive

When we test the `ShowContactsDirective` we only care about two test cases. One when the input evaluates to `true` and one when the input evaluates to `false`. The two cases are in Table 4.4 below:

Table 4.4: When the Show Contacts directive is set to true.

Test Case	Input	Displays
The element should be hidden when the input evaluates to `true`.	true	Element
The element should be hidden when the input evaluates to `false`.	false	Nothing

Now that we have seen the two test cases, let's create the test suite.

4.3.3 Setting up the Show Contacts directive test suite

To start, navigate to `/website/src/app/contacts/shared/show-contacts` and create a file called `show-contacts.directive.spec.ts`. First, we include the dependencies that the tests need. Add the following statements at the top of your file to import the Angular dependencies:

```
import { Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
```

These are the same classes that we imported for our Favorite Icon tests. Because we already covered those classes we won't cover them again. Now let's import two custom dependencies created for the Contacts application.

```
import { ShowContactsDirective } from './show-contacts.directive';
import { getElement } from '../../testing';
```

The `ShowContactsDirective`, is of course the code that we'll be testing. The `getElement` function is very similar to the `getStarElement` function that we used earlier – it's a small helper function that takes in a fixture and return the element that we want to test. If you would like to see how its implemented feel free to check out `/website/src/app/contacts/testing/get-element.ts`.

Now that we have our imports, we need create a `TestComponent` as we did for our attribute test. Go ahead and copy the below code under the `import` statements:

```
@Component({
  template: `
      <div *appShowContacts="true">
        <p>This is shown</p>
      </div>
      <div *appShowContacts="false">
        <p>This is hidden</p>
      </div>
    `
})
class TestComponent { }
```

You will notice that the first `<div>` contains `*appShowContacts="true"`. `appShowContacts` is the name that is used in the template for the `ShowContactsDirective`. You can think of `appShowContacts` as an alias for the `ShowContactsDirective`.

This will be the first element we check, and it should contain the child `<p>` element, because the `appShowContacts` is set to `true`. The second `<div>` contains `*appShowContacts="false"`. For this test, we want to check to see that this `<div>` does not contain the child `<p>` element, because the `appShowContacts` is set to `false`.

Now we create a `describe` function to house all of our tests. We also go ahead and add the fixture and `beforeEach` method. In the line below the `import` statements add the following code:

```
describe('Directive: ShowContactsDirective', () => {
  let fixture: ComponentFixture<any>;

  beforeEach(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [ShowContactsDirective, TestComponent]
    }).createComponent(TestComponent);
    fixture.detectChanges();
  });

  afterEach(() => { fixture = null; });
});
```

You may notice that this is almost exactly the same as the attribute directive tests. The only difference is we replaced `FavoriteIconDirective` with `ShowContactsDirective` because we are testing a different directive.

4.3.4 Creating the ShowContactsDirective tests

The two tests will follow the same format from before. We get the element we want to test from the fixture and then check the DOM to see if renders as it should. For our first test, we check if an element is rendered if the input is set to `true`. Add the following code above the `afterEach`:

```
it('should be displayed when the input evaluates to true.', () => {
  const element = getElement(fixture);
  expect(element.innerText).toContain('This is shown');
});
```

This test case will pass, because the child element with the text "`This is shown`" uses `appShowContacts=true` in our `TestComponent`. For our second test, we check that the content from the second div with the `appShowContacts` set to false does not show in the rendered HTML. Add the below code under the previous code:

```
it('should be hidden when the input evaluates to false.', () => {
  const element = getElement(fixture);
  expect(element.innerText).not.toContain('This is hidden');
});
```

Go ahead and fire up the terminal if you don't have it open and execute `ng test`. Both test should pass. You can check out the completed test is at `/chapter04/show-contacts.directive.spec.ts` for your reference. That's it for testing directives! In the next chapter, we'll look at testing another important concept in Angular, Pipes.

4.4 Summary

In this chapter looked we covered the following important topics:

- There are three types of directives in Angular: components, attribute and structure directives. They are all similar in that they both encapsulate reusable functionality. The difference between components and attribute and structure directives are that components have a view.
- Attribute directives can be used to change the appearance of an element and structure directives are used to add and remove elements to the DOM.
- The testing of attribute and structure directives are similar in that you will be setting the initial state of an element, perform the desired action, and then test to confirm that the expected change occurs.
- The `configureTestingModule` takes in an object that has to utilize the `TestModuleMetadata` interface. You can either create a variable that sets the type to `TestModuleMetadata` and the pass the variable into the `configureTestingModule`

method. Or create an object with the relevant configuration data and the pass it into the `configureTestingModule` method.

5

Testing pipes

This chapter covers:

- Testing pipes
- Understanding pure functions versus functions with side-effects
- Using the `transform` method

Often, you'll want to modify data that's displayed in a template. For example, you may want to format a number as a currency, or transform a date into a format that's easier to understand, or even simply make some text uppercase. In situations like these, Angular provides a way to transform data using something known as a *pipe*.

Pipes simply take input, transform that input, and then return some transformed value; because of this, writing tests for pipes is straightforward. Pipes depend only on their input; a function whose output depends on only the input passed to it is known as a *pure function*.

When a function can do something other than return a value, it's said to have a *side-effect*. A side-effect could be changing a global variable or making an HTTP call. Pure functions don't have side-effects. Pipes in Angular are pure functions, which is why they're so easy to test.

In this chapter, we'll cover everything you need to know to test pipes.

5.1 Introducing the PhoneNumberPipe

For the rest of the chapter we'll be testing a custom pipe called the `PhoneNumberPipe`. This pipe takes in a phone number as number or a string in valid format and then transforms it into a phone number format according to the user's specification. We need to write tests for the pipe so that we know that it transforms data correctly. Without tests we just have to hope that the pipe works correctly. If we don't test the pipe, we may not know how the pipe will handle invalid data, or we might be afraid to modify it without breaking it in some subtle way. Writing

tests lets us stop living on hope, and gives us confidence that we can improve our application without introducing bugs.

A pipe in Angular is a class that defines a `transform` method. This is the method responsible for formatting the pipe's input. The signature for the `transform` function looks like this:

```
transform(value: string, format?: string, countryCode?: string): string
```

`Value` is passed into the function from the left of the pipe. In our earlier example, `value` was the percentages in decimal format. The `format` is an optional string parameter that determines how the phone number is formatted. Different valid values for `format` are listed in table 5.1.

Table 5.1: Recognized format values

Number Separator Format	Phone Number Format
default	(XXX) XXX-XXXX
dots	XXX.XXX.XXXX
hyphens	XXX-XXX-XXXX

`countryCode` is another optional string parameter that adds a prefix to the phone number as an international county code. For example, if we pass in a `countryCode` of "`us`" (for the United States) and a format "`default`", the resulting phone number would be +1 (XXX) XXX-XXXX.

For simplicity, `PhoneNumberPipe` only works with phone numbers that follow the North American Numbering Plan (NANP). Thus, the available country codes are limited to the countries in the NANP. If you're curious about the acceptable country codes, look at the code in `country-dialing-codes.ts`. There's an object there that contains the two-character country abbreviation as a key and the international country code as the value.

Now that we know a bit about `PhoneNumberPipe` we can test it like so:

1. Set up the test dependencies.
2. Test the default behavior.
3. Test the `format` parameter.
4. Finally, test the `countryCode` parameter.

5.2 Testing the PhoneNumberPipe

We'll continue with testing the Contacts app, as we've done in previous chapters. If you need to set it up, follow the instructions in appendix A.

Open `website/src/app/contacts/shared/phone-number` and you should see the following files:

```
phone-number
```

```
    country-dialing-codes.ts
    index.ts
    phone-number-error-messages.ts
    phone-number.model.ts
    phone-number.pipe.ts
```

Table 5.2 describes each file.

Table 5.2 – Description of files

File	Description
`index.ts`	We use the index.ts file so that we can import the PhoneNumberPipe without using the complete file name. That way when you are trying to import PhoneNumberPipe you can use: `import { PhoneNumberPipe } from './phone-number.pipe';` Instead of the more verbose: `import { PhoneNumberPipe } from './phone-number.pipe/phone-number.pipe';` Notice the addition to the filename in bold. Using index.ts files like this is a common practice.
`country-dialing-codes.ts`	This file contains the country dialing codes that will be used by our PhoneNumber model.
`phone-number-error-messages.ts`	This file contains all the error messages that are used by the PhoneNumberPipe and PhoneNumber model.
`phone-number.model.ts`	This is the model that we will use to store data. The PhoneNumber model also contains the utility methods to transform the data.
`phone-number.pipe.ts`	This is the file that contains the PhoneNumberPipe.

Feel free to open each one of the files to get a feel for the source code that you'll be testing. When you're ready to move on, create a file named `phone-number.pipe.spec.ts` in the `phone-number` directory that will store our tests.

5.2.1 Testing the Default Usage for a pipe

First, let's test the default behavior of the `PhoneNumberPipe`. The following is an example of the default usage of the `PhoneNumberPipe`:

```
{{ 7035550123 | phoneNumber }}
```

There are two different cases that we cover the default usage of the `PhoneNumberPipe`, which are listed in table 5.3.

Table 5.3: Default Test Cases

Test Case	Number	Displays
A phone number that is a ten-character string or ten-digit number. should transform to the following (XXX) XXX-XXXX format.	7035550123	(703) 555-0123
Nothing will be displayed when a phone number that is not a ten-character string or ten-digit number.	703555012	

TESTING FOR A VALID PHONE NUMBER

Let's start off by testing the default usage if the phone number is valid. Copy the code for the first default test in listing 5.1 into the `phone-number.pipe.spec.ts` file that you just created.

Listing 5.1 First default test case

```
import { PhoneNumberPipe } from './phone-number.pipe'; (1)

describe('PhoneNumberPipe Tests', () => { (2)
  let phoneNumber: PhoneNumberPipe = null;

  beforeEach(() => { (3)
    phoneNumber = new PhoneNumberPipe();
  });

  describe('default behavior', () => { (4)
    it('should transform the string or number into the default phone format', () => {
      const testInputPhoneNumber = '7035550123';
      const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber);
      const expectedResult = '(703) 555-0123';

      expect(transformedPhoneNumber).toBe(expectedResult); (5)
    });
  });

  afterEach(() => { (6)
    phoneNumber = null;
  });
});
```

(1) Import the PhoneNumberPipe.
(2) Test suite for all of our tests.
(3) The setup part of our tests using beforeEach to set a new instance of the PhoneNumberPipe to the phoneNumber variable before each test.
(4) A nested, second test suite only for default behavior.
(5) The assertion where we expect a correct phone number to be formatted correctly.
(6) The teardown part of our test where we set the phoneNumber variable to null to destroy the reference.

Let's break this down by section:

```
import { PhoneNumberPipe } from './phone-number.pipe';
```

First off, we import all of the dependencies that our test needs. Because the pipe is a pure function, we don't need any of the Angular testing dependencies.

```
describe('PhoneNumberPipe Tests', () => {

});
```

We then add a `describe` function to house all our tests for the `PhoneNumberPipe`.

```
let phoneNumber: PhoneNumberPipe = null;

beforeEach(() => {
  phoneNumber = new PhoneNumberPipe();
});
```

Inside our test suite we need to create a global variable named `phoneNumber` that has a type of `PhoneNumberPipe` and is set to `null`. We use a `beforeEach` function so that a new instance of the `PhoneNumberPipe` is created before each test is executed.

```
describe('default behavior tests', () => {
  it('should transform the string or number into the default phone format', () => {
    const testInputPhoneNumber = '7035550123';
    const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber);
    const expectedResult = '(703) 555-0123';

    expect(transformedPhoneNumber).toBe(expectedResult);
  }));
});
```

Above, a `describe` block defines the nested test suite that contains our tests for default behavior. We declare our test input in the `testInputPhoneNumber` variable, save the transformed result in `transformedPhoneNumber` and set our expected result in the `expectedResult`. The assertion at the bottom of the test checks that the transformed phone number matches our expected result.

```
afterEach(() => {
    phoneNumber = null;
});
```

Finally, we'll add an `afterEach` function to ensure that the `phoneNumber` variable does not contain a reference to an instance of the `PhoneNumberPipe`. Run `npm test` and you should see output like figure 5.1:

```
13 06 2016 08:52:13.990:INFO [Chrome 51.0.2704 (Mac OS X 10.11.5)]: Connected on
 socket /#g0Euu8Bk3eZbin35AAAA with id 49045452
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 1 of 1 SUCCESS (0 secs / 0.007 sec
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 1 of 1 SUCCESS (0.009 secs / 0.007
 secs)
```

Figure 5.1: First successfully executed pipe test

That's it for our first test. The tests in the rest of the chapter follow the same format as the first test:

```
describe(describe a suite of tests, () => {
  it(describe the specific test case, () => {
    declare our test variables
    tranform the data
    expect(the transformed data).toBe(what we expect);
  });
});
```

Tests for pipes all follow this structure because, as mentioned before, pipes are pure functions. There's no need to mock or set anything up – we simply pass the function some input and confirm the result is what we'd expect.

TESTING THE PIPE WITH INVALID PHONE NUMBER

For the second test we'll verify that if the input number is not a ten-digit number, then nothing will be shown. Copy the `it` block in our previous test and paste it directly below your first test.

Change the descriptive text in the `it` block to "`should not display anything if the length is not ten digits`". Then change the `textInputPhoneNumber` to "`703555012`". Notice that the length of the new phone number is only nine digits long. Now, set the `expectedResult to ''`. We expected the result to be an empty string, because that's what should be returned if the phone number is invalid.

The completed test should look like the following code in listing 5.2.

Listing 5.2 Test for invalid phone number

```
it('should not display anything if the length is not ten digits', () => { (1)
  const testInputPhoneNumber = '703555012'; (2)
  const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber);
  const expectedResult = ''; (3)

  expect(transformedPhoneNumber).toBe(expectedResult);
});
```

(1) Update title of the test.
(2) Updated test input is an invalid phone number.
(3) Updated expected result is an empty string.

If you run ng test you will see something like the example in figure 5.2.

```
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 1 of 2 SUCCESS (0 secs / 0.007 sec
ERROR: 'The phone number you have entered is not
      the proper length. It should be 10 characters long.'
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 1 of 2 SUCCESS (0 secs / 0.007 sec
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 2 of 2 SUCCESS (0 secs / 0.009 sec
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 2 of 2 SUCCESS (0.013 secs / 0.009
 secs)
```

Figure 5.2: Two passing default behavior tests

Notice that the error message "`The phone number you have entered is not the proper length. It should be 10 characters long.`" is printed out to the console along with the successful test execution messages. This is expected because the `PhoneNumberPipe` throws an error message if the phone number is not 10 characters long. All console logging statements are printed out to the terminal when tests run.

Now that we've tested the default behavior, let's look at testing a pipe with a single parameter.

5.2.2 Testing Pipes with a Single Parameter

Sometimes, you'll need to change the behavior of a pipe by passing it a parameter. For example, you can change the format of the output of the `PhoneNumberPipe` by passing "`dots`", "`hyphens`", and "`default`" as a parameter.

Table 5.4 recaps the different options for the format parameter.

Table 5.4 – Test Cases for Format Parameter

Test Case	Format	Number	Displays
If "default" is used or no parameter is specified, then the number will be in.	default	7035550123	(703) 555-0123
If "dots" is passed in as a parameter then the number should be in XXX.XXX.XXXX format.	dots	7035550123	703.555.0123
If "hyphens" is passed in as a parameter then the number should be in XXX-XXX-XXXX format.	hyphens	7035550123	415-555-0122
If an unrecognized format is passed in as a parameter the default format, (XXX) XXX-XXXX should be used.	gibberish	7145550111	(714) 555-0111

An example usage of your Phone Number pipe with a single parameter is as follows:

```
{{ 7035550123 | phoneNumber | 'dots' }}
```

In this example, we pass "`dots`" as a parameter.

Now, that we understand how the second parameter is used for pipes, let's look at some tests. Add the code in listing 5.3 directly below the `describe` block that you created in listing 5.1.

Listing 5.3 Dots format test

```
describe('phone number format tests', () => { ❶
  it('should format the phone number using the dots format', () => {
    const testInputPhoneNumber = '7035550123';
    const format = 'dots'; ❷
    const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber, format); ❸
    const expectedResult = '703.555.0123';
```

```
    expect(transformedPhoneNumber).toBe(expectedResult);
  });
});
```

1. Test suite
2. The format type
3. Passing the format to our transform function

First off, notice that we have put this test inside of a test suite using a describe block. On the third line of the code, you we have a variable named `format` that is set to "`dots`". On the fourth line of the code, we pass that format variable in as a second parameter in our `transform` method. This is exactly how we test for the first parameter that is used by a pipe. By sending the *first* parameter into our `transform` method as the *second* parameter.

Run `ng test` and you should see output like that shown in figure 5.3.

```
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 1 of 3 SUCCESS (0 secs / 0.007 sec
ERROR: 'The phone number you have entered is not
      the proper length. It should be 10 characters long.'
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 1 of 3 SUCCESS (0 secs / 0.007 sec
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 2 of 3 SUCCESS (0 secs / 0.01 secs
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 3 of 3 SUCCESS (0 secs / 0.011 sec
Chrome 51.0.2704 (Mac OS X 10.11.5): Executed 3 of 3 SUCCESS (0.016 secs / 0.011
 secs)
```

Figure 5.3: Displays our three passing tests

Now that you understand how to test the first parameter, it's time for a little exercise.

EXERCISE

Create the three tests for the '`default`', '`hyphens`' and '`gibberish`' formats using the information provided in table 5.4 below the first parameter test that you just added in listing 5.3.

SOLUTION

All your tests should be very similar. The only thing different should be the new format type and the expected result based on that format type. Your three new tests should look like the following in listing 5.4.

Listing 5.4 Remaining format tests

```
it('should format the phone number using the hyphens format', () => {
  const testInputPhoneNumber = '7035550123';
  const format = 'default'; 1
  const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber, format);
  const expectedResult = '(703) 555-0123'; 2

  expect(transformedPhoneNumber).toBe(expectedResult);
```

```
});


it('should format the phone number using the hyphens format', () => {
  const testInputPhoneNumber = '7035550123';
  const format = 'hyphens'; (1)
  const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber, format);
  const expectedResult = '703-555-0123'; (2)

  expect(transformedPhoneNumber).toBe(expectedResult);
});

it('should format the phone number using the default format if unrecongized format is
       entered',() => {
  const testInputPhoneNumber = '7035550123';
  const format = 'gibberish'; (1)
  const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber, format);
  const expectedResult = '(703) 555-0123'; (2)

  expect(transformedPhoneNumber).toBe(expectedResult);
});
```

(1) **Format types.**
(2) **The expected formatted phone number.**

Run `ng test` and now you should see six passing tests. Now that we have one parameter tests handled, let's take a look at how to test multiple parameters.

5.2.3 Pipes with multiple parameters

Pipes can take multiple parameters if needed. Our `PhoneNumberPipe` has the ability to handle two parameters. So far, we have covered the first parameter and how it's responsible for formatting phone number to be formatted and the second parameter is the county code. Table 5.5 is the test cases for the country code parameter.

Table 5.5 – Test Cases for Country Code Parameter

Test Case	Number	Country Code	Displays
If "dots" is passed in as a parameter and the country code is correct then the number should be in XXX.XXX.XXXX format.	7035550123	us	+ 1 703.555.0123
If an unrecognized country code is passed in then there will be no country code applied.	7035550123	zz	703.555.0123

For simplicity, our `PhoneNumberPipe` only supports countries in the North American Numbering Plan. We need to test that ensures that each parameter is accepted and actually works as expected. Copy the code in listing 5.5 directly below the `describe` block that you created earlier that contains the phone number format tests.

Listing 5.5 Country code test

```
describe('country code parameter tests', () => {
  it('should add respective country code', () => {
    const testInputPhoneNumber = '7035550123';
    const format = 'default';
    const countryCode = 'us'; (1)
    const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber, format,
       countryCode); (2)
    const expectedResult = '+1 (703) 555-0123'; (3)

    expect(transformedPhoneNumber).toBe(expectedResult);
  });
});
```

(1) New variable that stores the country code.
(2) The countryCode variable is passed into the transform method as a third parameter.
(3) Expected result with country code.

This test is similar to the earlier tests for passing the first parameter to the pipe. The only difference is that before we were testing the *second* parameter, whereas now we're passing a *third* parameter to our transform method. You may be picking up on a pattern. So, if we want to test a *fourth* pipe parameter, then we would pass a value into the *fifth* parameter in our `transform` method. This pattern will continue for as many pipe parameters as we want to test.

EXERCISE

Write a test case that that when the country code is not recognized it only transforms the phone number format and does not add a telephone country code. Make sure that you run `ng test` to see if your test works as expected.

First Hint: You can just copy the above test and make modifications as necessary.

Second Hint: The country code should be a country code should be something not listed as a country in the North American Numbering Plan (https://en.wikipedia.org/wiki/North_American_Numbering_Plan).

SOLUTION

We need to change only two variables. The `countryCode` should be changed to something that is unrecognized. Then then `expectedResult` should just be the default format with no international country code prefixed the phone number. Your test should look something like listing 5.6.

Listing 5.6 Test for invalid country code

```
it('should not add anything if the country code is unrecongized', () => {
  const testInputPhoneNumber = '7035550123';
  const format = 'default';
  const countryCode = 'zz'; (1)
  const transformedPhoneNumber = phoneNumber.transform(testInputPhoneNumber, format,
```

```
      countryCode);
  const expectedResult = '(703) 555-0123'; ❷

  expect(transformedPhoneNumber).toBe(expectedResult);
});
```

❶ Unrecognized country code.
❷ The expected result without a country code.

Run `ng test`, if you haven't, and you now should see eight passing tests. If you have any issues, checkout the complete test at and look for any discrepancies. In the next chapter, we will start looking at testing services.

5.3 Summary

In this chapter, you learned the following:

- Because pipes simply take in a value as input, transform that value, then returns transformed input, testing pipes is straightforward because they're *pure functions*, which means there are no side-effects. *Side-effects* are changes that occur outside a function are that function is executed. A Common side-effect is the changing of a global variable.
- When we are testing pipes, we are mainly testing the `transform` method where we specify the different parameters that are to be used with our pipe.

6

Testing services

This chapter covers:

- Understanding the role services play in an Angular application
- Using dependency injection with service unit tests
- Using stubs for creating isolated unit tests
- Testing services that return results asynchronously using promises and RxJS observables
- Testing web services with Angular's HTTP utilities

In this chapter, we'll create and test the services we need for setting preferences in the Contacts app, and for loading and saving Contacts from a server. The `PreferencesService` will save application settings to the user's browser using either cookies or `localStorage`. The example shows how to test with both synchronous services and services that return promises. The `ContactService` uses the Angular `HTTP` client to fetch and store data from a REST service using *observables*, which are similar to promises but are designed for returning continuous streams of values. Both examples demonstrate you how to configure service tests, set up mocks and dependencies, and exercise service interfaces through test-driven development. By the end of this chapter, you'll know how to write and test services that use the `HTTP` client, and you'll learn how to move business logic into tested services to help organize your application's architecture.

6.1 What are services?

Angular services are, generally, the parts of your application that don't directly interact with the UI. Picture this: you are searching for pictures through an image service like Imgur. You type a search term, see a spinner pop up briefly, and then a set of images matching your search appear on the screen. What is happening during the time the spinner is running? It's

invisible, behind-the-scenes work performed by services in the application. What type of work? Often, it's saving or retrieving data. Other times, data is changed or created for use by the UI. Services can also be used as communication channels between application components.

Services provide a mechanism to write non-UI code in a way that is modular, reusable, and testable. As you'll see, code located in services is easier to understand and maintain compared with writing and maintaining the same functionality inside a UI component.

For the most part, Angular services do not modify the DOM or directly interact with the UI. Except for that, there's no limit on what functionality an Angular service can encompass. In well-designed applications, most of the application logic and I/O should be located inside a service. Any code creating UI elements or handling user input should be in a component.

Let's consider how we use services in the Contacts app. The app loads with a list of contacts. How do they get there? When `ContactListComponent` initializes, it asks `ContactService` for a list of contacts. `ContactService` looks to see if it has any contacts, and seeing that it doesn't, gives the HTTP client the API URL and key, and makes a GET request. The HTTP client handles making the request to the remote server, which responds with a list of contacts. The contacts get passed back to the `ContactService` and then back to the `ContactListComponent`. In this example, the `ContactService` knows the service address for getting contacts, but it doesn't know how to negotiate the network call, because that responsibility is owned by the HTTP client.

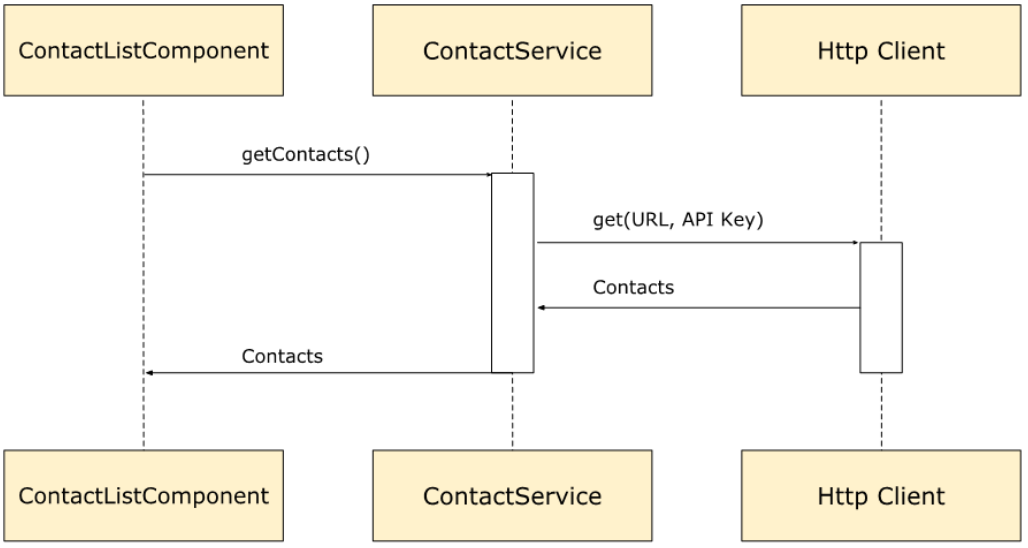

Figure 6.1. The ContactListComponent uses the ContactService, which itself uses the HTTP Client service.

Do I need services for my app?

It's possible, but not advisable, to write a complete Angular application without writing a single service. When you are prototyping an application or just trying to a get feature to work, it's often simpler to place all of a feature's code within the component that uses it. Over time, it becomes harder to maintain that code and even harder to share the work between components. That's when you need a service.

Let's consider an example using the Contacts application. While displaying a list of contacts, the user should be able to sort the list by first name, last name, or email address. To remember the sort order for the user's next visit, we save a value to the browser's local storage. The first time we write this code, it's easy to add to the `ContactList` component. But what happens when we need to save additional user preferences? And what if we need more flexibility for also saving the preferences to a web service?

As our requirements grow more complex, it makes sense to move this logic into a service. In fact, we may want to create two services: one to handle validating application preferences and another service for interacting with whatever storage medium we're using. Moving this logic into a service makes it easier for us to add preferences functionality into other areas of the application.

6.2 How do services work in Angular?

At the most basic level, Angular services are TypeScript objects. They're singletons, meaning that they're created once and can be used anywhere in the application.

Typically, Angular services implement the `@Injectable` class decorator. This decorator adds metadata that Angular uses for resolving dependencies. Angular uses the class itself to create a provider *token* that other components will use for defining provider dependencies.

A service is instantiated only once, and that instance is shared among components that define that service as a dependency. This technique reduces memory use and makes it possible for services to act as brokers for sharing data between components.

By now, it should be clear that services and Angular's dependency injection are closely related concepts. In fact, Angular offers many built-in services, including `Http` and `FormBuilder`. Many of the third-party libraries designed to work with Angular are also services. Before we start testing our services, we need to have a clearer understanding of what Angular's dependency injection does, because we can't write services without it.

6.2.1 Dependency injection

The key to understanding testing Angular services is to understand Angular's dependency injection system. Why do we need dependency injection in the first place? Isn't it good enough for us to be able to use `import` to pull other JavaScript libraries into our source code?

When we create new instances of our classes, it's possible that we don't know specific details about the dependencies our class needs. For example, suppose we create a dependency on a `Storage` service. If our service imports a specific storage mechanism, we are locked into using that one implementation and no others, when in fact we don't really care about a specific implementation of storage, only that it supports the methods we'll invoke when persisting data.

Dependency injection is a system that supplies instances of a dependency at the time we instantiate our own class. We don't need our service to do the work of importing and instantiating a dependency; the dependency injection system will do it for us. When the constructor of our service executes, it will receive an instance of a dependency already created by the dependency injection system, and the service will use the injected code instead of the imported class.

Listing 6.1 A service using Angular dependency injection

```
import { Injectable } from '@angular/core';
import { BrowserStorage } from './browser-storage.service';    (1)

@Injectable()
export class PreferencesService {

  constructor(private browserStorage: BrowserStorage ) { }    (2)

  public saveProperty(preference: IContactPreference) {
    this.browserStorage.setItem(preference.key, preference.value);    (3)
  }

  public getProperty(key: string) : any {
    return this.browserStorage.getItem(key);    (3)
  }
}
```

(1) Import the class so that we can use the token to define the dependency
(2) Angular dependency injection uses the service constructor to look up and supply dependencies
(3) PreferencesService uses injected services, not BrowserStorage directly.

Dependency injection adds a little bit of additional abstraction to software, and thus complexity, but it gives developers a lot more flexibility in extending the functionality of their software. As long as an additional service implements the same interface as an existing service, it’s possible to take advantage of it without having to modify any other code. This is an especially helpful feature when unit testing because it helps us test in small, separate units while controlling the side effects of our software.

The bottom line is that dependency injection gives us the flexibility to write code that is not tightly coupled to other code. In other words, it lets us write code against an *interface* instead of an *implementation*. Angular's dependency injection helps us develop better code, and it's one of the features that makes Angular a great framework for organizing large applications.

Dependency-injection Tokens

You've probably noticed that we are importing the classes that are dependencies for the service we're testing, only to replace the implementation of those dependencies with our own mock objects. Why are we doing this? It's because Angular's dependency injection uses the class type as the *token* which becomes the key for its internal *token-provider* map.

When services are defined with dependencies, the service provides a copy of the token to Angular's injection system. Angular uses that token to look up the corresponding provider and returns it to the service.

What about cases where you want to provide just a string or object instead of a service function? It's possible to use a string as a token instead by using `InjectionToken` function from *@angular/core*. See the Angular documentation for further details (https://angular.io/guide/dependency-injection).

6.2.2 The @Injectable() class decorator

As you may recall, a decorator is a TypeScript feature adds some properties or behavior to a class or method. Angular includes a decorator for services, `@Injectable()`, which is a convenient way to mark your service as a class that can be used as a provider for Angular's dependency injection system. The decoration informs Angular that the service itself has its own dependencies that require resolution. Although it's possible to manually define your service as a provider the circumstances where you would want to do so are rare (unless you're using straight JavaScript ES5, in which case it's unavoidable).

A service, just like a `@Component()`, is able to define its own dependencies as well. A common example is any service that uses `HttpModule` for communicating with external services. As applications grow in complexity, it becomes normal for there to be several layers of services. Separating code into modular units helps promote reuse and it makes it easier to maintain the code. In Angular, services are designed to support reusability.

Is the `@Injectable()` decorator required for Angular services? Actually, no. If your service has no dependencies of its own, you can get by without marking the service as `@Injectable()`. A service without dependencies can be unit tested without needing the Angular `TestBed` or any Angular testing utilities. That's right, you can use simple unit tests! Before heading down that road keep this in mind: if you anticipate adding dependencies in the future, you may as well unit test your code with Angular so that your unit tests will require less refactoring.

Now that we have some background on Angular services and dependency injection, it's time to create our first service. Although it's a straightforward matter of creating services by hand, we'll use `angular-cli` because it also sets up basic unit tests for us as well.

6.3 Creating services with angular-cli

We will be using `angular-cli` to create services, although you can also create them by hand. The advantage of using `angular-cli` is that it automatically generates a basic service and corresponding test file that provides the boilerplate code for the Angular `TestBed`.

To create a service using `angular-cli`, run the following command from your project directory in your command line:

```
$ ng generate service my.service.name
```

This command creates the following two files:

```
my.service.name.ts
my.service.name.spec.ts
```

After these services are created, `angular-cli` produces this message:

```
Warning: Service is generated but not provided, it must be provided to be used
```

Never fear, this is just `angular-cli` reminding us that services need to be added to the `provider` metadata property of a component or module in order to be used. Where you include it depends on how whether the service is local to a component or used throughout the whole module.

The unit test file generated by `angular-cli` (which ends with `spec.ts`) is a very basic test that does nothing more than ensure that our service exists. Fortunately, this is all we need to get started. Even better, `angular-cli` also prepares the boilerplate code that configures `TestBed`, which (as previous chapters explain) sets up Angular for use with unit testing.

Listing 6.2 Basic service test spec generated by Angular-CLI

```
import { TestBed, inject } from '@angular/core/testing';
import { ContactService } from './contact.service';

describe('ContactService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({ ❶
      providers: [ContactService]
    });
  });

❷
  it('should ...', inject([ContactService], (service: ContactService) => {
    expect(service).toBeTruthy();
  }));
});
```

❶ TestBed is configured with the ContactService before everty test
❷ The first test is already configured and ready to run

Now we've learned the basics of using angular-cli to set up our services and tests. Before we jump into writing service tests, let's make sure we understand why we're writing this service in the first place.

6.4 Testing PreferencesService

The first service we'll create and test is `PreferencesService`, which we will use to store the user's last sort order on the *ContactsList* table. This service will take a value and save it to the browser's built-in storage system. When the app starts, it will use this value to set the sort state for the `ContactsList` table to be in the same order that the user last used it, even if they've refreshed the page or restarted the browser. This adds user experience polish to our

application that keeps the user from having to manually reset the sort order each time they open the app.

Before we start, we might wonder whether we even need to create a service. We could instead write the logic for persisting user preferences in the *ContactsList* component. After all, it seems simple to write settings directly to the user's browser. That's true, and it might even be the right choice if our task is to create a rapid prototype rather than build a production application. But there are good reasons to split this functionality out of the component and into a service. Let's look at some of them.

Whenever we start using a browser feature like `localStorage`, it isn't long before we start discovering requirements that weren't obvious at first. For example:

- Validating key names and values
- Preventing naming conflicts with other preference keys
- Ensuring the app has a fallback mechanism such as using browser cookies
- Limiting the size of the storage used by our application
- Checking that storage is available

Few of these things seem obvious when your only goal is to use a browser's built-in storage, but they are important for a production application. Suppose you write this logic in your component: how do you make use of it when you need to save additional information to the browser's storage? If you copy and paste your code, it will start to become difficult to remember to change it in every place whenever you need to fix a bug or add a feature. The way to solve this challenge is to write your code once in a service and then use that service wherever you need its functionality.

Now that we know why we need a service, let's start creating it.

WRITING THE PREFERENCESSERVICE

Begin by using `angular-cli` to create the preference service. Run the following command at the prompt:

```
$ ng generate service PreferencesService
```

This command creates the files `preferences.service.ts` and `preferences.service.spec.ts`. Remember that the application reminds us that the service must be provided to be used. Before writing tests, let's add the `PreferenceService` provider to the application's `AppModule` (listing 6.3).

Listing 6.3 Adding the newly created service to app.module.ts

```
@NgModule({
    {…} 1
    providers: [
        BrowserStorage,
        ContactService,
        PreferencesService,
        PreferencesAsyncService
```

```
      ],
    })
export class AppModule { }
```

❶ Declaration and imports are hidden in this figure

Now that the service is created, we can start setting up the unit test framework. The basic work is already done by `angular-cli`, so we can proceed with setting up the parts we need to test `PreferencesService`. Add the code in listing 6.3 to `preferences.service.spec.ts`.

Listing 6.4 Setting up the unit tests for PreferencesService

```
import { TestBed, inject } from '@angular/core/testing';
import { IContactPreference, PreferencesService } from './preferences.service';

describe('PreferencesService', () => {

    beforeEach(() => {    //❶
        TestBed.configureTestingModule({
            providers: [PreferencesService]
        });
    });

    it('should create the Preferences Service', inject( //❷
        [PreferencesService], (service: PreferencesService) => {
    expect(service).toBeTruthy();
    }));
});
```

❶ The TestBed module is configured before every test
❷ The first test only checks that the service test setup is right

Now that we know our basic unit test is working, we need to consider how our service will persist the data. Are we going to use `localStorage`, cookies, or some other browser API? We don't know right now, so let's pretend that we have the information by creating a fake service called `BrowserStorage` that only allows us to avoid relying on any specific storage implementation. This extra indirection lets us continue writing `PreferencesService` without having to solve the storage problem.

To use this technique, we'll create an extremely simple service that doesn't implement any logic—this service only exists to provide a token and a simple storage interface. Later on, when we're ready to use a real service implementation, we can expand `BrowserStorage` to connect to the real persistence service. For now, we will place this code in `browser-storage.service.ts`. It only has two methods, `getItem` and `setItem`, and they don't need to do anything other than define their input and output types.

Listing 6.5 The BrowserStroage interface

```
@Injectable()
export class BrowserStorage {
```

```
  getItem: (property: string) => string | object; (1)
  setItem: (property: string, value: string | object) => void;
}
```

(1) This interface will be used for testing storage

We can use this interface to create a `BrowserStorageMock` for our test. A *mock* is an object that substitutes for a real service. With our mock, we'll define both the `getItem` and `setItem` methods, and these will only ever be called from within the unit test. Within individual unit tests, we will use *spies*. A spy is a function that invisibly wraps a method and lets us control what values it returns or monitor how it was called. By using the token from `BrowserStorage` and by supplying the same methods, we can use our mock for unit testing instead of relying on the real implementation. We only need to configure `TestBed` to use `BrowserStorageMock` whenever a service calls for `BrowserStorage` as a dependency.

Now, let's look at the `PreferencesService` class – it only has two methods: `saveProperty()` and `getProperty()`. The `saveProperty()` method takes a `ContactPreference` object and saves it for later retrieval. The `getProperty()` method takes the property name and returns the saved value. In this chapter, we only look at the unit test for saving the property, but you can see all unit tests in this book's project repository. The `saveProperty` method takes one argument, an object which represents the name and value of the preference item we're saving. For consistency, we'll create a TypeScript interface (`IContactPreferences`) to describe that object. The method uses the object and writes that value to the `browserStorage` service (remember that browserStorage helps us separate the read and write operations from the actual persistence implementation).

Listing 6.6 The saveProperty method

```
interface IContactPreference {
  key: string;
  value: string | object;
}

public saveProperty(preference: IContactPreference) {
    if (!(preferences.key && preference.key.length)) {
        throw new Error('saveProperty requires a non-blank property name');
    }
    this.browserStorage.setItem(preference.key, preference.value);
}
```

The first test ensures that `saveProperty` works correctly. This function's responsibility is to receive a `ContactPreference` object and store it. To test it, we need to make sure the method receives a valid argument and calls the appropriate storage class. To write this test, we'll spy on the `browserStorage` service, invoke `saveProperty` with a valid argument, and then verify that our expectations are met.

You'll notice that we're using a spy on the `browserStorage` service. From there on, we can check to see how many times the method was invoked and with what parameters. In this test,

we're checking to ensure that the `setItem` method was called with the expected parameters. Spies are not unique to Jasmine—most testing frameworks support spies. You can learn more about Jasmine spies in the Jasmine documentation at http://jasmine.github.io.

Listing 6.7 Testing the saveProperty method

```
import { TestBed, inject } from '@angular/core/testing';
import { PreferencesService } from './preferences.service';
import { BrowserStorage } from "./browser-storage.service";

import { logging } from "selenium-webdriver";
import Preferences = logging.Preferences;

class BrowserStorageMock {    (1)
    getItem = (property: string) => ({ key: 'testProp', value: 'testValue '});
    setItem = ({ key: key, value: value }) => {};
}

describe('PreferencesService', () => {

    beforeEach(() => {
        TestBed.configureTestingModule({
            providers: [PreferencesService, {    (2)
                provide: BrowserStorage, useClass: BrowserStorageMock
            }]
        });

    });

    describe('save preferences', () => {

        it('should save a preference', inject([PreferencesService, BrowserStorage],
            (service: PreferencesService, browserStorage: BrowserStorageMock) => {    (3)

                spyOn(browserStorage, 'setItem').and.callThrough();    (4)
                service.saveProperty({ key: 'myProperty', value: 'myValue' });
                expect(browserStorage.setItem).toHaveBeenCalledWith('myProperty', 'myValue');
      (5)
            })
        );
    });
});
```

(1) Create BrowserStorageMock
(2) Configure the TestBed dependency injection to use BrowserStorageMock instead of the real service
(3) Use inject to get the BrowserStorageMock
(4) Add a spy to browserStroage.setItem
(5) Check the spy to make sure it was called from saveProperty()

Let's review our work so far.

1. We started by using Angular-CLI to generate a service and its associated unit test file.
2. We've added the service to the `AppModule`.
3. Then we wrote a basic unit that verifies the service and test file are setup correctly.

4. Next, we expanded the unit test setup by creating a mock for the `BrowserStorage` service.
5. We created a unit test that checks for the existence of `saveProperty`.
6. After that, we created a unit test that verifies the behavior of `saveProperty` when called with a valid parameter.

What else remains? You might have noticed we only tested the "happy path," that is, the condition where we used the method correctly. To ensure that our code responds appropriately with bad input, we'll need to add additional tests.

6.4.1 Testing for failures

What if something goes wrong? The input to our function could be bad. Maybe there are missing parameters, or maybe the parameters are of the wrong type (for example, we expected an array and received a number). Or maybe the input looks good, but the values don't pass the validation checks. It's likely that multiple things could go wrong, and it's good to have a plan for responding to the misuse of your code.

Testing for failures works much the same way as testing the happy path. We'll set up the tests the same way and invoke the method the same way, but instead we provide incorrect inputs to insure our code responds correctly.

In the following example, we call saveProperty() using the incorrect parameter of not having a value for the key name. What we expect to happen is that the service will throw an error. Not having a key might not break our code, but it could cause problems with the persistence solution we end up using. To catch the problem early, our code should throw an error if the key is empty or not defined.

Normally, if a function in a test is executed and throws an error, it causes the whole test to fail. Because this test is specifically ensuring that the function throws an exception, Jasmine needs more setup. To solve this puzzle, the test needs to define a function that itself calls the function that will throw. By providing this function to Jasmine in the expect block, and then using the `toThrowError()` matcher, Jasmine will execute the function and anticipate that an error will be thrown as a result. After the error is thrown, we can verifiy that the error value has the correct type and error message. For the purposes of this test, it's sufficient to check that an error was thrown.

Listing 6.8 Unit test for checking that a bad input throws an error.

```
it('saveProperty should require a non-zero length key',
    inject([PreferencesService],  (service: PreferencesService) => {

    const fn = () => service.saveProperty({ key: '', value: 'foo' }); ❶

    expect(fn).toThrowError(); ❷
  })
);
```

❶ Create a wrapper for any function that is supposed to throw an error

❷ Expect the function to throw an error.

Remember that the Jasmine methods for testing if errors are thrown require that the `expect` parameter be a function. This is different from other tests, so make sure you aren't calling your test function inside the `expect` method!

What we've learned in this chapter works well for services that operate synchronously. We used the example of synchronous persistence, and this pattern works well whenever you want two services to work together without directly coupling them. However, we often need to handle asynchronous events such as remote service calls or user interactions. Testing asynchronous services is a bit more complex.

6.5 Testing services with promises

As you've been following along in this chapter, you may have wanted to ask, "What if I want to use some type of web service to save my preferences?" This is a great question: the way we've designed the *ContactPreferences* service so far depends on having a synchronous persistence media. In this section, we will create an alternate Preferences service with asynchronous methods that return promises, thereby giving us more flexibility to change the underlying implementation. (For an example of asynchronous testing with observables, see section 6.6.)

> **NOTE** If you haven't worked with promises before, all you need to know for now is that a promise provides a way to write asynchronous JavaScript without nesting multiple levels of callbacks. Promises became a standard JavaScript feature in ES2015. You can learn more about promises at the Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises).

Services that respond to method calls asynchronously need to be tested differently because Jasmine needs to know when to end an individual test. A Jasmine test will automatically complete if it's not specifically told to wait. The second parameter to a Jasmine `it` block is a function that takes an optional callback parameter, usually called `done`. If this parameter is supplied by a test, Jasmine will wait for `done` to be called before ending the test.

> **BEWARE** If you write a test with an `expect` inside an async block without letting Jasmine know that it's an async test, Jasmine will report the test as passing! Beware of this behavior, and recall that the practice of *red-green-refactor* helps us to be aware of these types of oversights. If your test skipped red and went straight to green then there might be an error in your code!

If you've ever tried spent a lot of time in Jasmine trying to figure out why your asynchronous test was passing when it obviously should be failing, you aren't alone. If this is your first time using Jasmine, then this is something you'll be glad to know right from the beginning. The correct way to write an asynchronous unit test in Jasmine is to pass the `done` callback to the unit test and invoke it when you want the test to end.

The following code sample illustrates a common mistake in an asynchronous test—forgetting to supply and invoke the `done` callback argument from the `it` method. Later we will see that Angular supplies some helpers so we don't need the `done` method, but for basic asynchronous tests in Jasmine, follow this example.

Listing 6.9 Incorrect and correct way to write asynchronous Jasmine tests

```
it('is an asynchronous test', () => { ❶
    setTimeout(() => {
        expect(true).toBe(false);
 });
});


it('is an asynchronous test', (done) => { ❷
    setTimeout(() => {
        expect(true).toBe(false);
     done();
 });
});
```

❶ Unit test unexpectedly passes because done callback not defined or called.
❷ Unit test fails because it doesn't complete until the done callback is invoked.

This method of writing asynchronous tests will always work with Angular unit tests, but when you are writing tests that need to `inject` dependencies into the test block, this syntax isn't easy to use because it requires accessing the dependencies by exposing them with a `beforeEach` block, and this can make it less clear which tests require which injected services. Fortunately, Angular gives us more options for easily writing tests with asynchronous logic.

The evolving Angular asynchronous testing story

Angular has a way of detecting changes that happen when asynchronous JavaScript is called. This system, called a *Zone*, changes the way asynchronous browser APIs like `setTimeout` work. When Angular was revealed to the public, developers needed to understand how Zones worked to write unit tests, making it especially difficult for developers new to Angular to write unit tests.

The Angular team was responsive to the challenge of writing async tests, adding support to *Zone.js* to make it easier to write tests. The first API made available to developers was `async/whenStable`. Running a test in an `async` zone made it much easier to write tests for code that changed the DOM. After changing the application state, a developer could wait for the `fixture` to signal when it was safe to continue a test. The `whenStable` method returns a promise, making it fairly easy to write tests that worked correctly with asynchronous changes.

Later, another API became available, `fakeAsync/tick`, that makes it easier to write tests in a more synchronous style. Tests written inside a `fakeAsync` block can pretend to "fast forward" asynchronous events by calling `tick()`.

When writing async tests, should you use `done`, `async/whenStable`, or `fakeAsync/tick`? It's really your choice, but in this chapter, we'll be using `fakeAsync/tick` because it's newer and the Angular team seems to prefer it.

Although there are several ways of invoking asynchronous JavaScript, including timeouts, DOM events, generator functions, and so on, our async re-implimentation of PreferencesService will use promises. In the previous section, we looked at `setProperty`, so in this example we'll look at the asynchronous version of `getProperty`.

A promise constructor has two parameters, both of which are callbacks that resolve the promise. If everything goes well, the `resolve` callback is invoked to return a value, but if something goes wrong, the `reject` callback is used to signal that an error occurred. Because `BrowserStorageAsync` returns promises already, we will return that promise with no modification unless `getPropertyAsync` is called with an invalid parameter, in which case we'll return a new rejected promise with an appropriate error message.

Listing 6.10 PreferencesAsync Service

```
import { Injectable } from '@angular/core';
import { BrowserStorageAsync } from './browser-storage.service';
import { IContactPreference } from './preferences.service';

@Injectable()
export class PreferencesAsyncService {

  constructor(private browserStorage: BrowserStorageAsync) { }

    getPropertyAsync(key: string) : Promise<IContactPreference> {
        if (!key.length) {
            return Promise.reject('getPropertyAsync requires a property name');    (1)
        } else {
            return this.browserStorage.getItem(key);                               (2)
        }
    }
}
```

(1) Reject with an error message if no key is passed
(2) Otherwise return the promise from BrowserStroageAsync

This test looks very similar to our first example, but this time we need to import additional methods from `@angular/core/testing`, namely `fakeAsync` and `tick`.

Writing the asynchronous test requires additional setup. First, we need to import the asynchronous testing utilities from the core Angular library. Then, `BrowserStorageAsyncMock` needs to return values that are promises, just like the real implementation.

The fakeAsync helper lets us write tests that appear to be asynchronous, but which are actually synchronous. This is helpful because it reduces the amount of boilerplate in the unit test, resulting in tests that are easier to write and understand. The `fakeAsync` helper wraps around `inject` (if we need it), so that we don't need to call `done` throughout as we would in a normal Jasmine unit test.

As we test the `getPropertyAsync`, which returns a promise, we will assign the values from the promise to locally-scoped values. It's also a good practice write a `catch` handler to ensure that no unexpected error was thrown.

After the promise, we call `flushMicrotasks` to process pending promises. Finally, we check for the received values to match the expected values.

Listing 6.11 Importing inject and fakeAsync from @angular/core/testing

```
import { TestBed, async, fakeAsync, flushMicrotasks, inject } from '@angular/core/testing';
       (1)

import { BrowserStorageAsync } from "./browser-storage.service";
import { PreferencesAsyncService } from './preferences-async.service';

class BrowserStorageAsyncMock {    (2)
    getItem = (property: string) => Promise.resolve({ key: 'testProp', value: 'testValue '});
    setItem = ({ key: key, value: value }) => Promise.resolve(true);
}
describe('PreferencesAsyncService', () => {
    beforeEach(() => {
        TestBed.configureTestingModule({
            providers: [PreferencesAsyncService, {
                provide: BrowserStorageAsync, useClass: BrowserStorageAsyncMock
            }]
        });
    });

    it('should get a value', fakeAsync(inject([PreferencesAsyncService, BrowserStorageAsync],
        (service: PreferencesAsyncService, browserStorage: BrowserStorageAsyncMock) => {
            spyOn(browserStorage, 'getItem').and.callThrough();

            let results, error;

            service.getPropertyAsync('testProp')  (3)
                .then(val => results = val)
                .catch(err => error = err);

            flushMicrotasks();    (4)

            expect(results.key).toEqual('testProp');
            expect(results.value).toEqual('testValue');
            expect(error).toBeUndefined();  (5)
           expect(browserStorage.getItem).toHaveBeenCalledWith('testProp');
        }))
    );
});
```

(1) Importing asynchronous testing methods
(2) Mocking the asynchronous service response
(3) Invoking the promise and assigning the results
(4) Process the promise microtasks
(5) Ensure the error value was not assigned

To test the `getPropertyAsync` method, we assumed our persistent storage service would return a promise, which we provided as a mock. In the unit test, we assigned the output of the promise to local values (results, error) and then checked our expected values.

Remember that your Angular services don't need to call flushMicrotasks—this is a testing-only helper that makes it easier to test asynchronous services. We're using it here to make sure the promises resolve before checking our expected values. By using `fakeAsync`, you don't need to call Jasmine's `done` to end your asynchronous test.

In the next section, we'll cover asynchronous unit testing for expected failures.

6.5.1 Testing for failures with asynchronous services

Testing for failures in services that use promises is similar to what we covered in section 6.4.1 for testing synchronous errors. One major difference is that promises have a different way of resolving errors.

When something goes wrong when calling a promise, the `reject` callback of the promise will be called with any error information, and then the `catch` method which resolves the promise can handle any error processing. Promises work by calling either the `resolve` or `reject` callbacks, thus throwing a new Error will not work. The `reject` callback must be used.

Listing 6.12 Testing for rejected promises

```
it('should throw an error if the key is missing', fakeAsync(inject([PreferencesAsyncService],
       (service: PreferencesAsyncService) => {
        let result, error;
        service.getPropertyAsync('')          (1)
            .then(value => result = value)    (2)
            .catch((err) => error = err);     (3)

        flushMicrotasks();
        expect(result).toBeUndefined();       (4)
        expect(error).toEqual('getPropertyAsync requires a property name');  (5)

    }))
);
```

(1) Call getPropertyAsync with an invalid value
(2) Use BrowserStorageAsyncMock default return value
(3) Catch the expected error and assign it locally
(4) We shouldn't get a preference value back
(5) We should get an error with this error message

Remember to write tests that cover all of your expected paths into and out of your functions. Writing asynchronous code is easier than ever with promises, but it's still important to write robust code that checks for correct input values and raises errors when problems arise.

Our asynchronous examples so far have used promises, but in the next section we will write unit tests that handle RxJS-based observables. Angular deeply incorporates RxJS in its design, especially in the Angular HTTP client. This type of code requires special setup, and so we will cover an example in depth.

6.6 Testing HTTP services with observables

In this section, we'll cover one of the most common uses of Angular services: connecting an application to a remote API using the Angular `HTTP` client. We'll also cover unit testing services that return an RxJS observable, since this is the default response type for the `HTTP` client.

Testing HTTP services can be tricky because they are written to access live web services, but making a network call from our unit tests would break their isolation. If our unit tests aren't isolated, it's much more difficult to pinpoint a failure in the system under test.

Here are some other reasons we shouldn't call web services directly from our tests:

- The computer running the unit tests may need to make calls over a network, adding complexity to the test configuration.
- There's no way to guarantee the same result each run of the test.
- We might not have control over the service, which may mean we get back different results if the service specification changes.
- The service itself may be discontinued or moved to a different address.
- We can't control the response times of servers, leading us to set long timeouts to try to anticipate how quickly our calls will return.

It seems like we're in a difficult position. We need to test that our service is making the correct calls to the `HTTP` client, and we also need to test that our service correctly responds to both successes and failures. How can we do this without running into the difficulties listed above?

We'll need to do two things to prepare unit tests for services that make HTTP calls:

- Prevent the `HTTP` client from making a real network call.
- Fake a response to the `HTTP` client with a value we supply.

Fortunately, Angular has anticipated this problem and gives us an Angular HTTP client that makes unit testing much easier.

Recall when we were testing `PreferencesService` that we created a fake storage service so that we could isolate our test from the browser's real storage system. This is a common pattern in unit testing, and we'll be using it again to test our web service. When testing with the `HTTP` client, our setup is a little bit more complicated because we need to configure mock dependencies for the `HTTP` client itself.

6.6.1 Mocking the `Http` Client Backend

Before writing any test that uses the Angular HTTP client, we need to do some extensive configuration to emulate the behavior of our network service calls.

Because the setup for this test is more complicated than the other examples in this chapter, we'll present the code example in two parts - first the TestBed setup, and then the unit test itself.

When configuring the test bed for these unit tests, we need to define the service we're testing and provide settings for the HTTP client. The HTTP client requires two injected objects: a class that implements the `ConnectionBackend` class, and a `RequestOptions` object for providing options when making a request. The `ConnectionBackend` class provides the mechanics for how an HTTP request will communicate to a service. Usually, this is `XMLHttpRequest`, but in a test we can use the Angular `MockBackend` utility.

To override the behavior of the HTTP client, we'll use `MockBackend`, which prevents any requests from being sent over the network and also provides some utility methods for checking the status of our requests (we'll look at these later).

The TestBed configuration will be the same for every test regardless of whether we're fetching or sending data. Each type of method under test requires additional setup.

Listing 6.13 Setting up TestBed before each test

```
import { TestBed, fakeAsync, inject } from '@angular/core/testing';
import { Http, BaseRequestOptions, Response, ResponseOptions, RequestMethod } from
       '@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';

import { ContactService } from './contact.service';

describe('ContactsService', () => {
    beforeEach(() => {
        TestBed.configureTestingModule({
            providers: [
                ContactService,
                MockBackend,
                BaseRequestOptions, (1)
                {
                    provide: Http,
                    useFactory: (backend, options) => new Http(backend, options), (2)
                    deps: [MockBackend, BaseRequestOptions]
                }
            ]
        });
    });

});
```

(1) Configure the dependnecies for BaseRequestOptions
(2) Configure the Http service by using the MockBackend

Now that we've set up TestBed, we can be sure that calling any method on the HTTP client will hit the MockBackend instead of accidentally calling through to a real service. We can proceed with the additional configuration for the unit tests.

`ContactService` makes different calls to the web service for working with Contacts data. Each of these calls is based on an HTTP verb (such as GET, POST, DELETE). For each of these operations, we'll need to set up our tests to provide an appropriate response.

Before each test, we can use `inject` to get the `MockBackend` from Angular's dependency injection system, and then save a reference to it so that we can configure the request response as we need it for each test. We'll also generate a new response for each test run - doing so will help maintain isolation between each unit test. The response we create is simple: the data property contains an array of a single contact. The 'body' property of `mockGetContactsResponse` should conform to the model we expect the live service to return to our application.

> **NOTE** Although we are creating a fake server response, this technique exposes one of the difficulties of writing these types of tests: we still have a hidden dependency on the server itself. We're making a leap of faith that the server response will match what our code does. One way to navigate this thorny situation is to write a contract for the service using a specification language such as OpenAPI, and then use those specifications as an input for your unit tests. That goes beyond the scope of this chapter, but it's worth investigating if you're writing code to access web services.

After each test, we'll also check the instance of the `mockBackend` to ensure there are no pending or unresolved connections to the backend. If there are any, the test will show an error.

Each method in our service (e.g., `getContact`, `setContact`) needs to set up the unit test differently to account for different types of server responses. The response from a `GET` request will be different from that of a `POST`, for example.

Once all of this setup is complete, we can write the body of the test. At the top of the unit test, we'll subscribe to the `connections` observable from the `mockBackend`. Each unit test makes a backend call, and for each call we ensure that it's the right type (e.g., `GET` for the `getContacts` method), and then tell the connection to respond with our mock response object.

After all that setup, we'll finally get to the actual test portion of the unit test. The `ContactService.getContacts` method returns an observable. In the unit test, we'll subscribe to the observable and test the values that are emitted from it. We expect that calling the service will return an array of Contacts, and that the first contact will equal our test data.

Listing 6.14 Unit testing ContactsService.getContacts()

```
describe('getContacts', () => {
  let mockBackend;
  let mockContact;
  let mockGetContactsResponse;

  beforeEach(inject([MockBackend], (_MockBackend_) => {
    mockBackend = _MockBackend_; (1)
    mockContact = { id: 100, name: 'Erin Dee', email: 'edee@example.com' };
    mockGetContactsResponse = new Response(new ResponseOptions({ body: { data: [ mockContact
        ] }})); (2)
  }));

  afterEach(() => { (3)
```

```
    mockBackend.verifyNoPendingRequests();
    mockBackend.resolveAllConnections();
  });

  it('should GET a list of contacts',
    fakeAsync(inject([ContactService], (service) => {

      mockBackend.connections // 4
        .subscribe((connection: MockConnection) => {
          expect(connection.request.method).toEqual(RequestMethod.Get);
          connection.mockRespond(mockGetContactsResponse);
        });

      service.getContacts() // 5
        .subscribe((contacts) => {
          expect(contacts[0]).toEqual(mockContact);
        });
    }))
  );
});
```

1. Get a reference to MockBackend
2. Create a mock response with mock contact details
3. Make sure there are not pending requests
4. Configure the mock backend with the mock response
5. Execute getContacts and test the results

As you can see, there's a lot to testing services that use the Angular HTTP client. Let's look at Figure 6.2 to see the relationship of the dependencies and how we setup the mocks for the test. ContactService has a dependency on the HTTP client. In the setup, we configure Http with MockBackend and BaseRequestOptions. Later, we create a new Response with a ResponseOptions instance created from our mockContact.

We wait for the mockBackend to receive a request, and we respond to that request with the mockGetContactsResponse, which is what appears to be the response from the server, and hence, the HTTP client. We then use that value for testing our response.

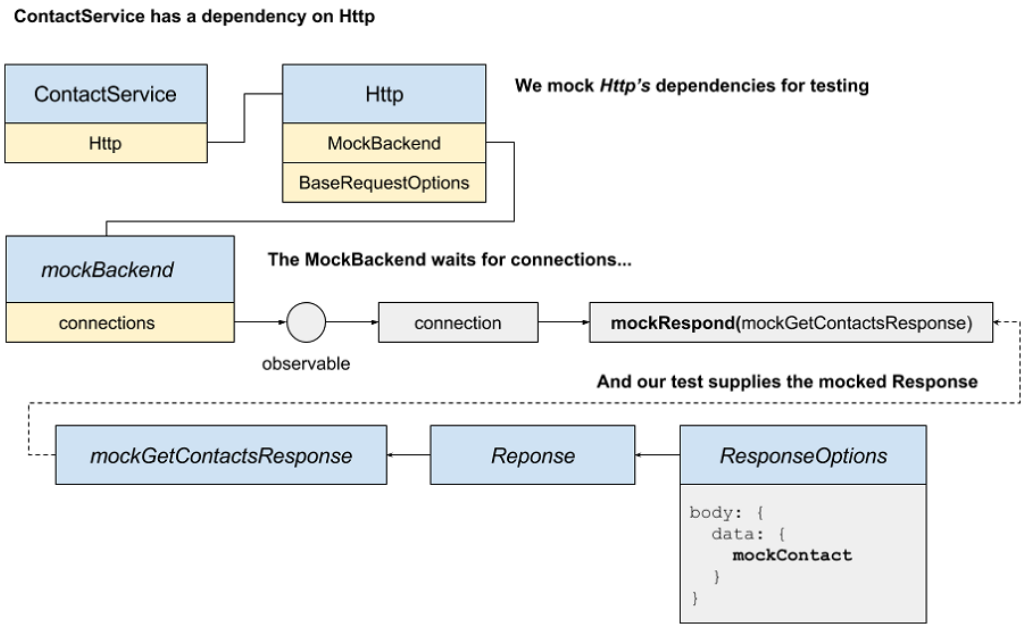

Figure 6.2 Creating the mocks for testing with the HTTP Client

6.7 Summary

In this chapter, we covered testing services in Angular and learned about:

- Angular dependency injection, and why it's important to understand it for setting up and writing unit tests.
- Using services in Angular to write modular code where each module does only one thing, but can work with other modules (i.e., dependencies) to accomplish a larger task.
- Substituting service dependencies in unit tests so that our code is always interacting with systems in our control, and how we can spy on these interactions to insure our code is working properly.
- The differences between unit testing synchronous and asynchronous code, as well as the specific case of testing RxJS observables.

7

Testing router

This chapter covers:

- An overview of the router and its role in single-page applications
- Configuring the router for an Angular application
- Testing components that use the router
- Testing advanced router configuration options

No man is an island, and no Angular application is a solitary component. Almost every Angular application needs a way to convert the web address in the location bar to some destination in the web application, and this work is performed by the router.

In simple applications, such as the Contacts application used throughout this book, the router configuration is as simple as associating a URL path with a component, but the router is capable of more-advanced functionality as well. For example, your application may have sections that are accessible only if a user has permission to see the data, and the router can verify the user's credentials before even loading that part of the application.

In this chapter, you will learn more about the router, how to configure the router, and some examples of testing both the router code and components that need to use the router. Understanding how to test the router in your application builds on skills you've already learned for testing components and services, so it's recommended that you understand the chapters on both components and services before continuing with the material in this chapter.

7.1 What is the Angular router?

The Angular router is a part of the Angular framework that converts the web address to a specific view of the Angular application, which makes the router an integral part of the Angular application architecture. In practice, all Angular applications need to define a router, which

makes the router an essential part of Angular development. When a user navigates to a URL for an Angular application, the router parses the URL and determines the appropriate component and data to load. This means that whenever a URL changes, whether by direct entry or by clicking a link, it's the router that does the work of setting up the appropriate application state. Each segment of the path encodes information about your application's state. The Angular router examines the path and breaks it into a series of tokens that are used for loading components and for making data available to those components.

Suppose the Contacts application is available at `www.example.com`, and a user wants to edit the contact corresponding to ID 5. The page could be reached at `http://www.example.com/app/contacts/5/edit`. Figure 7.1 shows how the website URL corresponds to the router configuration. First, notice that the base path, which is configured in the HTML of your application using the `<base href>` tag, is not included as part of the router configuration. Angular uses the base path as the starting point (or default view) of the application. All the other URLs in the application will be under the base path. Second, notice that the dynamic portion of the URL, the `contactId`, is special in that it is preceded with a colon. The name of this label will be used to send this parameter along to any components that need to use it.

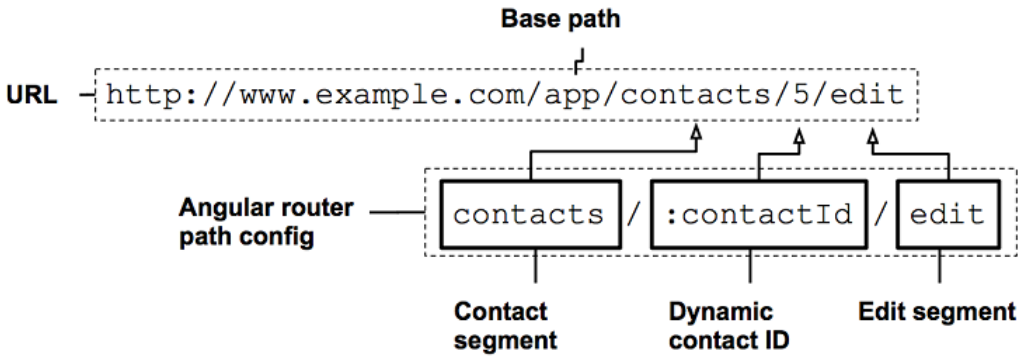

Figure 7.1 How the application URL corresponds to the router configuration

The router configuration could be very simple for an application with only a handful of routes (such is the case with the Contacts application), or it could be a very long configuration, spread among many different files (as may be the case for a large enterprise application). Every time a route is changed, several route guards (lifecycle hooks that run during a navigation attempt) are executed. Whenever a new route is loaded, the components on that route have access to the route's parameters, which those components may require for further loading and displaying data, or for resolving user's actions as they perform tasks in the application. The router configuration affects how applications will function throughout an

application, therefore it is useful to add tests to validate components that interact with it, or to test the route guards that the router itself fires.

There are two different means of testing code related to the router. The first is by testing how components receive values from the router or call actions on the router. The second is by testing the application code that is call from the router's route guards.

Imagine the router from a component's point of view: a component may need route information passed into it, or it may need to tell the router to perform navigation actions. When writing tests from this approach, you'll use a fake router configuration (the components won't care that it's fake) and write tests to ensure the components are interacting correctly.

Now think of testing from the router's point of view. Suppose an unauthorized user is trying to access the app's administration panel? In this case, you'll want to use a router configuration that checks to see if the user is logged in and has the right roles or permissions before the user is allowed to continue. In this case, the router configuration is more important because it drives which route guards are called.

Router and component interactions are tested following the same pattern of testing any other parts of Angular, and what you've already learned about testing components and services applies. The only difference is that these tests will use the Angular *RouterTestingModule*, a built-in testing utility that was created for these scenarios.

In either case, you'll need to have a router configuration in order to test your code. The next section looks at this topic.

7.1.1 Configuring the router

A router configuration is mandatory in using the router. During testing, you have the option of using your application's routes configuration, but most of the time you'll create a configuration specifically for testing so that you have more control over your tests.

The most basic router configuration does no more than associate a URL path to a component. Listing 7.1 shows the router configuration for the Contacts application, which includes the default "home" page, pages for adding and modifying contact entries, and a default for an unknown route. In some of the path entries, you'll see part of the path prefixed with a colon (for example, `'edit/:id'`). The colon prefix tells the router that the value in that portion of the path is dynamic, and that the router should make that data available to all the components. With a configuration this simple, there is little worth testing because there's no behavior to test. The only reason why there might be value in writing a test is to provide an extra layer of control when changing any of the path information, but we don't recommend writing tests unless the route configuration is more complex, such as when using router route guards.

7.1 Router configuration for Contacts application

```
export const routes: Routes = [
  { path: '', component: ContactsComponent },                    ❶
  { path: 'add', component: NewContactComponent },
  { path: 'contacts', component: ContactsComponent },
```

```
  { path: 'contact/:id', component: ContactDetailComponent },    ❷
  { path: 'edit/:id', component: ContactEditComponent },         ❷
  { path: '**', component: PageNotFoundComponent }               ❸
];
```

❶ Default application route which is the "home page" of the application
❷ Routes with dynamic path sections
❸ Default component for when the path doesn't match the route configuration

Typically, testable code related to the router concerns some type of state change, such as attempting to navigate to a route or by loading a component following a transition (including initial load). In a test, the router configuration needs only the relevant properties configured as are required for the test itself. Because of this, it's unusual to use your application's live route configuration for unit testing. More often, your unit tests will only contain as much route information as it takes to execute a single group of tests.

7.1.2 Route guards: the router's lifecycle hooks

Route guards are services that implement an interface for methods that run when the router attempts to load a route. Each time a route change occurs in an Angular application, the router will execute any lifecycle hooks configured for that route. Route guards make it easy to coordinate application behavior as the user moves from place to place in the application. Functions written for a route guard will have a signature that receives the current route configuration or the component associated with the current route, and that function will return a Boolean value, or a promise or observable which returns a Boolean. The router uses this value to determine whether it can continue to load the route, or if it should abort.

The order of these events is as follows:

- *CanDeactivate*

  Runs before a user can even leave the current route. This is useful for prompting the user if they have any unsaved forms or other unfinished activities. If this

- *CanActivateChild*

  For any route that has child routes defined, this hook runs first. This is useful if a feature has some sections that are restricted to users based on roles or permissions.

- *CanActivate*

  This hook must return `true` for the route to continue loading. Like with `CanActivateChild`, this hook is useful for restricting unauthorized users from loading application features.

- *Resolve*

  If a user is allowed to activate the attempted route, the `resolve` method is used to load data prior to activating the route itself. The data is then available from the `ActivatedRoute` service in the routed components.

In addition to these hooks, there is also the `CanLoad` guard, useful in applications which use `lazy loading`, a method of dynamically loading only the parts of an application relevant to a user's needs.

This wraps up the introduction of the Angular router, router configuration, and route guards. The next section covers testing your application code related to the router.

7.2 Testing routed components

In this section, you'll see two different examples of tests related to interactions between components and the router. The first example is a component that makes dynamic calls to the router service. The test will check the calls to the router to ensure the dynamic paths are being created correctly. The second example is a component that receives parameters passed from the `ActivatedRoute` service.

Because the Contacts application does not make use of more-advanced routing features, the examples used in this chapter will be standalone code, but you will still be able to run the examples on your computer.

7.2.1 Testing router navigation with RouterTestingModule

Suppose you have a component that dynamically generates a navigation menu. Your test finds the menu element and clicks it. You expect the next route to load with the correct parameters, but clicking the link changes the URL and breaks Karma. How do we work around this problem?

Whenever you're testing a component that could cause a navigation event (as we are here by clicking on a link), use the `RouterTestingModule` to prevent Angular from loading the navigation target component. The `RouterTestingModule` intercepts navigation attempts and provides a means to check the parameters of the navigation attempt. Could you do this manually by providing your own mock router service? Of course, but it's easier to use the helper provided by Angular. In this section, the component being tested generates links dynamically based on a menu configuration, and the test uses the `RouterTestingModule` to verify that the link is working and that the target is correct.

> **WHAT IS ACTUALLY BEING TESTED HERE?** It's important to separate the ideas of testing a component and testing a router configuration. It's possible to test a component together with your application's route configuration, but every component defined by the routes must be imported into the test. In practice, this leads to fragile tests, because tests will break whenever the route configuration is changed or updated. In essence, using your production route configuration for testing places your entire app under test! We recommend not coupling your component tests to your application's route configuration. Instead, write your components so that the router configuration is passed to the component as a dependency (see chapter 6, Services) so that the route value can be provided as a mock value in testing.

GENERATE LINKS

The `NagivationMenu` component under test in listing 7.2 is a simple example purely for illustrating how to test components that interact with the router. If you were writing this component for production, it would have other behaviors, perhaps animation or nested menus. Remember that in this example, it's the behavior of the component with respect to the router, and not the router configuration itself, that's under test.

In this example, when the component initializes (`ngOnInit`), it receives its configuration from a service containing route information, which it uses to generate a list of links using the `routerLink` component. The test for this component ensures that the links are generated correctly and link to the expected targets.

7.2 The NavigationMenu component

```
@Component({      (1)
  selector: `navigation-menu`,     (1)
  template: `<div><a *ngFor="let item of menu" [id]="item.label" [routerLink]="item.path">{{
       item.label }}</a></div>`,     (1)
})     (1)
class NavigationMenu implements OnInit {    (1)
  menu: any;      (1)
  constructor(private navConfig: NavConfigService) { }    (1)
  ngOnInit() {    (1)
    this.menu = this.navConfig.menu;    (1)
  }    (1)
}     (1)
```

(1) The NavigationMenu component generates a list of links based on a configuration service

CONFIGURE ROUTES AND CREATE TEST COMPONENTS

Although the `NavigationMenu` component is simple, tests using the `RouterTestingModule` require a little bit of setup. The route configuration needs at least two routes configured: the initial route, and a second route to be the target of the navigation attempt. The initial route loads the component under test, and the second route doesn't matter, but it should be a valid target for whatever link the component constructs.

It's helpful to create a simple component for the test that only exists to be the target for the test. You could import another component in your application for this setup, but defining a simple target component in your test reduces the complexity and thus the number of things that could go wrong.

7.3 Creating test components for NavigationMenu test

```
@Component({     (1)
  selector: 'app-root',    (1)
  template: `<router-outlet></router-outlet>`,    (1)
})    (1)
class AppComponent { }    (1)
```

```
@Component({    (2)
  selector: `simple-component`,    (2)
  template: `simple`               (2)
})    (2)
class SimpleComponent { }          (2)
```

(1) The AppComponent is the test fixture in which the component is tested.
(2) This simple component will stand in as the target component in the test route configuration.

SET UP ROUTES

The last necessary setup is to import the `RouterTestingModule`, which will spy on navigation calls and make the results of such calls available for checking in the tests. The `RouterTestingModule` takes an optional route configuration.

Instead of using the application's route configuration, it's better to create a fake route configuration so that you don't need to import and configure all of the components into your test. You should make the test as simple as possible by avoiding pulling in unnecessary dependencies. For this test, you'll need two routes: the default route, which loads the component under test, and the target route.

Before each test, we configure `TestBed` with modules and mocks needed for the tests themselves. We run this prior to each test so that the values created by the tests are reset between each instance, which prevents the tests from interacting with one another. To avoid the repetition in each test, the router will load the initial page and then advance the Angular application to settle any asynchronous events.

When a navigation event occurs, its resolves asynchronously, and this must be accounted for in the test. This example uses the Angular `fakeAsync` helper to handle settling asynchronous calls. When using `fakeAsync`, outstanding asynchronous calls must be resolved manually with the `flush()` method, and then the fixture needs to be updated with `detectChanges()`. Since these need to be called so often, this test defines a helper function called `advance()`, which makes the test code a bit easier to read.

7.4 Setup code to run before each test

```
let router: Router;
let location: Location;

let fixture;
let router: Router;
let location: Location;

beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [RouterTestingModule.withRoutes([ (1)
      { path: '', component: NavigationMenu }, (1)
        { path: 'target/:id', component: SimpleComponent } (1)
      ])],
    providers: [{
```

```
      provide: NavConfigService,
      useValue: { menu: [{ label: 'Home', path: '/target/fakeId' }] }
    }],
    declarations: [NavigationMenu, SimpleComponent, AppComponent],
  });
});

beforeEach(fakeAsync(() => {
  router = TestBed.get(Router);
  location = TestBed.get(Location);
  fixture = TestBed.createComponent(AppComponent);
  router.navigateByUrl('/'); ❷
  advance(); ❷
}));

function advance(): void { ❸
  flush(); ❸
  fixture.detectChanges(); ❸
} ❸
```

❶ Configure the RouterTestingModule with fake testing routes
❷ Start each test by navigating to the default route
❸ A test helper for fakeAsync that resolves and detects asynchronous side effects

There's a lot of setup for testing this component, so let's do a quick review:

- The component under test generates navigation links.
- The setup creates two mock components to facilitate the test, one for the app fixture and one for the target.
- The `TestBed` configuration uses `RouterTestingModule` with fake route information. Before each test, the default route is loaded and the fixture is updated.

We can now write the first test, which is pretty simple. After the fixture is set up, the NavigationMenu should generate links based on its input. The test gets a copy to a link, clicks it, and then checks with the Location service to see if the path updated to the expected target.

7.5 Testing generated NavigationMenu links

```
it('Tries to route to a page', fakeAsync(() => {
  const menu = fixture.debugElement.query(By.css('a')); ❶
  menu.triggerEventHandler('click', { button: 0 }); ❷
  advance(); ❸
  expect(location.path()).toEqual('/target/fakeId'); // ❹
})); ❶
```

❶ Get reference to a generated link element
❷ Send the link a click event
❸ Process the navigation attempt and update the fixture
❹ Test that the router location updated to the expected target

Why did it take so much setup for such a small test? Remember that the router is tightly integrated into the backbone of an Angular application. Because of that, it takes extra work to isolate the router from its configuration and from any side effects caused by navigating to different routes.

> **NOTE**: Although this test uses DOM elements and click events to activate the test, you can use this same technique to test components that call `Router.navigate()`. The setup is the same, but in the test, you'd trigger whatever event that would cause the navigation method to fire.

In the next section, we will cover another case of testing interactions between a component and the router, but where the component receives values from the router.

7.2.2 Testing router parameters

Deep linking is the ability to link to a view of specific content in a website or a web application. A URL for a deep link embeds information about the content (usually through an identifier), sorting and filtering parameters, and sometimes pagination parameters. For example, a car sales web application makes it possible to deeply link into a specific model, year range, sorted by price. That link can be saved and shared, and although the content is generated dynamically, the parameters are always the same.

TESTING ACTIVATEDROUTE COMPONENTS

In an Angular application, deep linking is implemented though route parameters. These parameters are captured by the router and made available to any component that requires them through the `ActivatedRoute` service. Whenever a user navigates to a different route, the `ActivatedRoute` service makes information about the route change available to components that use the service. One of the most basic uses of `ActivatedRoute` is to pass along a unique identifier for further content lookup. For example, in the Contacts application, the ContactEdit component uses `ActivatedRoute` to get the identifier for a contact. When the user navigates to `http://localhost/edit/1`, the router compares the path to the router configuration and extracts the last part of the path to be used as the value of `id`. Thereafter, this value is published to `ActivatedRoute` and all subscribing components are sent the update.

The test in this section will use a simplified example of how to test a component that has a dependency on `ActivatedRoute`. The values published by ActivatedRoute can either be subscribed to as an observable or as a *snapshot*, an object holding the last updated values for all parameters. Subscribing to an observable is a good choice for a long-lived component that needs to update regularly based on route changes (see chapter 6.6, Testing Http services with observables). Using the snapshot is simpler, and a good choice for when a component only needs to use route parameters upon construction. Testing setup for either is similar, but this example will use a snapshot.

The example in listing 7.6 is simplified for illustration purposes. Normally, a component for editing data would have a form and controls for modifying and saving the data, but here we are focusing on just loading the Contact ID from the `ActivatedRoute` service and using it in the template.

7.6 Simplified ContactEdit component using ActivatedRoute

```
@Component({
   selector: `contact-edit`,
   template: `<div class="contact-id">{{ contactId }}</div>`, (1)
})
class ContactEditComponent implements OnInit {
private contactId: number;
constructor(private activatedRoute: ActivatedRoute) { } (2)
ngOnInit () {
  this.contactId = this.activatedRoute.snapshot.params['id']; (3)
}
}
```

(1) Shortened template for illustration purposes
(2) The ActivatedRoute service is injected during construction
(3) The Contact ID is assigned on initialization

SETTING UP THE TEST

The component under test is simple. Compared with testing components that cause navigation events to occur, setting up the test for `ActivatedRoute` is much simpler. This component only listens for data and then renders its template.

The only mock this test requires is a mock for ActivatedRoute. TestBed will provide the mock value to the component. Notice that this test, unlike the `NavigationMenu` component test, doesn't use the `RouterTestingModule`. It isn't necessary for this test because no navigation is occurring.

7.7 Setting up the ActivatedRoute mock for component testing

```
let fixture;
const mockActivatedRoute = { (1)
  snapshot: { (1)
    params: { (1)
      id: 'aMockId' (1)
    } (1)
  } (1)
}; (1)

 beforeEach(() => {
   TestBed.configureTestingModule({
    providers: [
      { provide: ActivatedRoute, useValue: mockActivatedRoute} (2)
    ],
    declarations: [ContactEditComponent],
  });
});
```

```
beforeEach(async(() => { ❸
  fixture = TestBed.createComponent(ContactEditComponent); ❸
  fixture.detectChanges(); ❸

})); ❸
```

❶ The mock is the snapshot that the ActivatedRoute would generate as part of a route event.
❷ The mock ActivatedRoute is injected as the value for ActivatedRoute.
❸ The TestBed asynchronously initializes the TestBed fixture with the component under test.

TESTING THE COMPONENT

When the router resolves a navigation event, the `ActivatedRoute` produces a snapshot of the route data associated with the component at the point in time when the component is instantiated. This test doesn't use the router to generate this snapshot because the role of the `ActivatedRoute` in this test is to merely supply the snapshot. Supplying the snapshot here avoids the extra setup work that would be required to have the router generate the snapshot automatically. As long as you know what the snapshot looks like, you can use a mock instead, which simplifies the test.

From here, the test is a matter of initializing the component and checking the result. This is another asynchronous test. The values from the `ActivatedRoute` are made available to the component within the `ngOnInit` method. This process happens asynchronously, and this test is easier to read by using the Angular `async` test helper instead of the `fakeAsync` helper. This test checks the value of the Contact ID after it's rendered in the form to ensure that it's the value that's coming from the `ActivatedRoute` service.

7.8 Testing the ContactEdit component loading route parameters

```
it('Tries to route to a page', async(() => { ❶
  let testEl = fixture.debugElement.query(By.css('div')); ❷
  expect(testEl.nativeElement.textContent).toEqual('aMockId'); ❸
}));
```

❶ The test is wrapped in the async helper because the test is waiting for the component to initialize.
❷ A reference to the DOM node where the Contact ID should be rendered.
❸ Verify the template is rendered with the Contact ID from the ActivatedRoute

Because this component uses the `ActivatedRoute` snapshot, setting up the test is easy. If your component uses properties of `ActivatedRoute` that emit observables, then your mock would be an observable emitting the mocked properties.

7.9 Using a mock Observable for ActivatedRoute

```
const paramsMock = Observable.create((observer) => { ❶
  observer.next({ ❶
    id: 'aMockId' ❶
  });  ❶
```

```
  observer.complete(); ❶
}); ❶

beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        { provide: ActivatedRoute, useValue: { params: paramsMock }} ❷
      ],
      declarations: [ContactEditComponent],
    });
  });
```

❶ Creating an observable that will be used for testing ActivatedRoute params.
❷ The observable is used as the params method for the injected mock service.

USING A MOCK OBSERVABLE

Whether you use a route snapshot or create an observable for your mock `ActivatedRoute`, testing a component that reads values from `ActivatedRoute` is straightforward.

This is all you need to know about testing routes from the point of view of a component. The rest of this chapter will look at testing special features of routes that are especially useful in an enterprise single-page application—route guards and resolved data services.

7.3 Testing advanced routes

The router offers functionality that enables Angular applications to implement enterprise application-level features. With the router, it's easy to route unauthenticated users to a login page, pre-fetch data before loading components, and to lazy-load other modules to help reduce application start time. The router configuration can define hooks called *route guards* that are called prior to navigation events. Another feature of the router is pre-loading, or *resolving,* data before activating components.

7.3.1 Route guards

Enterprise web applications differ from other web applications in that enterprise applications have user authentication, user roles, privileges, and other means of allowing or preventing users from accessing features. The Angular router makes it easy to add some of this functionality to a web application. Route guards are specialized services that are run prior to a router navigation event. If they return *true*, the navigation can continue. Otherwise, the navigation attempt fails.

Why would you want to use a route guard instead of adding this functionality directly to a component? Route guards exist outside of your component; therefore, they let you separate the functionality of access or permissions from the core functionality of the component. Also, by defining route-guard services separately from components, re-using the validation logic becomes a matter of configuring your route configuration rather than adding additional component logic.

There are two categories of route guards: guards that check before a user tries to leave a current route, and guards that check before a user can load a new route. There is actually a third type of route guard, called a *resolve* guard, but that will be covered in the next section.

In this example (listing 7.9), you'll see a `CanActivate` route guard. To use this route guard, a route configuration entry specifies a new property called `canActivate`, which takes an array of route guards. The system under test in this example is `AuthenticationGuard`, a route guard that depends on the `UserAuthentication` service to see if a theoretical user is allowed to access the route. If not, the navigation attempt fails.

7.10 AuthenticationGuard service

```
@Injectable() (1)
class AuthenticationGuard implements CanActivate {     (1)
  constructor(private userAuth: UserAuthentication) {} (1)
  canActivate(): Promise<boolean> { (1)
    return new Promise((resolve) =>  (1)
       resolve(this.userAuth.getAuthenticated());      (1)
    );(1)
  }   (1)
}     (1)

@Injectable() (2)
class UserAuthentication { (2)
  private isUserAuthenticated: boolean = false; (2)
  authenticateUser() {     (2)
    this.isUserAuthenticated = true; (2)
  }   (2)
  getAuthenticated() {     (2)
    return this.isUserAuthenticated; (2)
  }   (2)
}     (2)
```

(1) The route guard implementing the CanActivate interface, which is the focus of the test.
(2) A fake service used to demonstrate separating the responsibility of the route guard from the user authentication service.

We've chosen to show `UserAuthentication` as a separate service. This makes the example a little more complicated, but it's good to reinforce the idea that a service that handles user authentication—which in real life would make network calls and have other complexities—should be separated from the route guard service itself. Smaller services are easier to test, and with this level of separation, it's easy to mock the dependencies for the system under test.

Setting up the test is similar to the setup for the `NavigationMenu` test. This test requires a component for initializing the application fixture and a simple component to act as the target for the navigation attempt. Listing 7.3 shows you the creation of these test components. What is new in this test setup? First, notice that this test once again uses the `RouterTestingModule` with a configuration specific to testing `AuthenticationGuard`. As mentioned before, it's better to create a test router configuration because it's less complicated and more reliable than

trying to use the application's router configuration. The test configuration specifies the `canActivate` property, which activates the code that is the focus of this test.

7.11 Setting up the AuthenticationGuard test

```
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [RouterTestingModule.withRoutes([        (1)
      { path: '', component: AppComponent },
      {
        path: 'protected',
        component: TargetComponent,
        canActivate: [AuthenticationGuard],           (2)
      }
      ])],
    providers: [AuthenticationGuard, UserAuthentication],
    declarations: [TargetComponent, AppComponent],
  });

  router = TestBed.get(Router);
  location = TestBed.get(Location);
  userAuthService = TestBed.get(UserAuthentication); (3)
});

beforeEach(fakeAsync(() => {
  fixture = TestBed.createComponent(AppComponent);
  router.initialNavigation();
}));
```

(1) **Using the RouterTestingModule**
(2) **Specifying the route guard for the test.**
(3) **This will be used to allow the authentication check to pass.**

The `AuthenticationGuard` will check the `UserAuthentication` service when the route navigation attempt occurs, so we capture a reference to the service in order to control it during the test. The testing setup is finished, so what remains are two tests. Both will attempt to navigate to the protected route. The first will make the attempt without authenticating, and in the second we'll manually authenticate the user. After the navigation attempt, the tests check the `Location` service to ensure the expected result.

7.12 AuthenticationGuard tests

```
it('tries to route to a page without authentication', fakeAsync(() => {
  router.navigate(['protected']);     (1)
  flush();
  expect(location.path()).toEqual('/');
}));

it('tries to route to a page after authentication', fakeAsync(() => {
  userAuthService.authenticateUser(); (2)
  router.navigate(['protected']);
  flush();
  expect(location.path()).toEqual('/protected');
```

```
}));
```

1. First, attempt to navigate to the protected route before authentication.
2. Next, authenticate first, and try the attempt again. This time it works.

These tests ensure that the route guards behave correctly under different application scenarios. There are other approaches you could use for testing route guards. For example, you could spy on the `canActivate` method to ensure that it's called as expected and that it's returning the correct response. Both methods work, so use whichever you prefer.

7.3.2 Resolving data before loading a route

There are times when you'll want to load data prior to activating a component. The *resolve* route guard specifies an object of key-value pairs where the keys are the name of data properties and the values are route guard services that fetch the data. Once all services have resolved, the data is made available to components through the `ActivatedRoute` service on the data property.

7.13 Configured resolver router guard

```
{
  path: 'contacts',
  component: TargetComponent,
  resolve: { (1)
    userPreferences: UserPreferencesResolver, (1)
    contacts: ContactsResolver (1)
  } (1)
}
```

1. A route configuration using a resolver route guard

Testing a resolver route guard uses the techniques already covered in section 7.3, and so we won't be providing a separate example. To test these resolvers, follow the same process as the `canActivate` route guard. The resolvers themselves usually interact with some other data service, for which you'll want to provide mock services. Then in the unit test, you'll inject the `ActivatedRoute` service and check that the values available on `ActivatedRoute.snapshot.data` match your expected values.

7.4 Summary

In this chapter, we covered testing the Angular router via components and route guard services where we explored the following:

- The purpose of the Angular router, and how it helps you write enterprise single-page applications.
- How components can use the router to navigate to other areas of the applications.
- How components can receive route parameters and resolved data from the router.

- Using route guards to block a user from navigating until some condition has been met, and how to test route guards.

8

Getting started with Protractor

This chapter covers:

- Understanding how Protractor works
- Writing your first Protractor test
- Interacting with elements
- Interacting with a list of elements
- Organizing tests with page objects

In the first part of the book, we saw how to create hermetic unit tests that verify our application's features work as expected in isolation. Unfortunately, just having a good suite of unit tests isn't enough to ensure that our application will behave as expected. Because unit tests validate the contact list web application in isolation, they can't verify that external services or dependencies work together with the application.

We could test both the workflow and external dependencies with a set of tests that interact with the contact list web application like a real-world user. We could just manually test the application; however, it's better to have automated tests. We call tests that interact with the application like a real user *end-to-end tests*. These tests launch a browser, navigate to the contact list web application, and interact with it like a real-world user. This kind of test is expensive to run, so instead of exhaustively testing every possible real-world user scenario, we'll pick a set of tests that cover only the most important scenarios.

It's also important to decide which scenarios were already covered by unit tests. The testing pyramid shown in figure 8.1 shows a good balance of test coverage. A Google Testing Blog post suggested a ratio of 70% unit tests, 20% integration tests, and 10% end-to-end tests. Because integration tests involve validating external dependencies, we group them together with end-to-end tests. End-to-end and integration tests are equivalent for our

purposes – integration tests just use mock versions of external dependencies (an in-memory database, for example), and end-to-end tests use the real version.

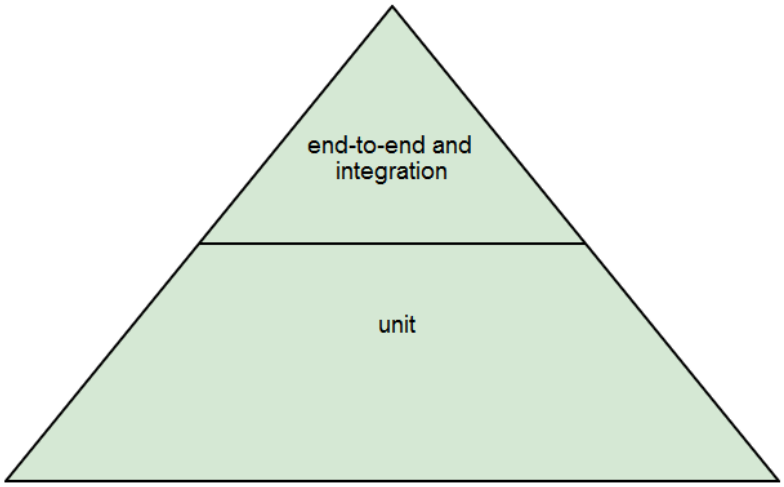

Figure 8.1: Testing Pyramid: unit tests and end-to-end and integration tests

There are two ways to write end-to-end tests: with selenium-webdriver library or with Angular's Protractor test framework. If you have previously written a selenium-webdriver test, you might have found that your tests become a nested chain of promises to synchronize browser commands. In contrast, Protractor wraps the selenium-webdriver APIs to make these asynchronous commands appear synchronous. Because Protractor wraps the selenium-webdriver library, it exposes the same set of APIs as selenium-webdriver. When writing end-to-end tests for Angular, it's better to use Protractor instead of selenium-webdriver.

Before creating a Protractor test suite, we'll need to cover the basics. This chapter will demonstrate how Protractor works and how to write your first Protractor test. We'll create a simple test that interacts with web elements on the screen. Finally, we'll show how to organize test code with *page objects*. When refactoring your test suite, using page objects helps reduce code duplication and creates maintainable code.

If you're already familiar with Protractor, feel free to skip to chapter 9 where we discuss timeouts and chapter 10 where we will cover advanced Protractor topics.

8.1 How Protractor works

Let's look at the big picture of how Protractor helps you write your browser tests. Protractor tests run on NodeJS and send commands to a selenium server. The selenium server is controls the browser via WebDriver commands and JavaScript functions (figure 8.2).

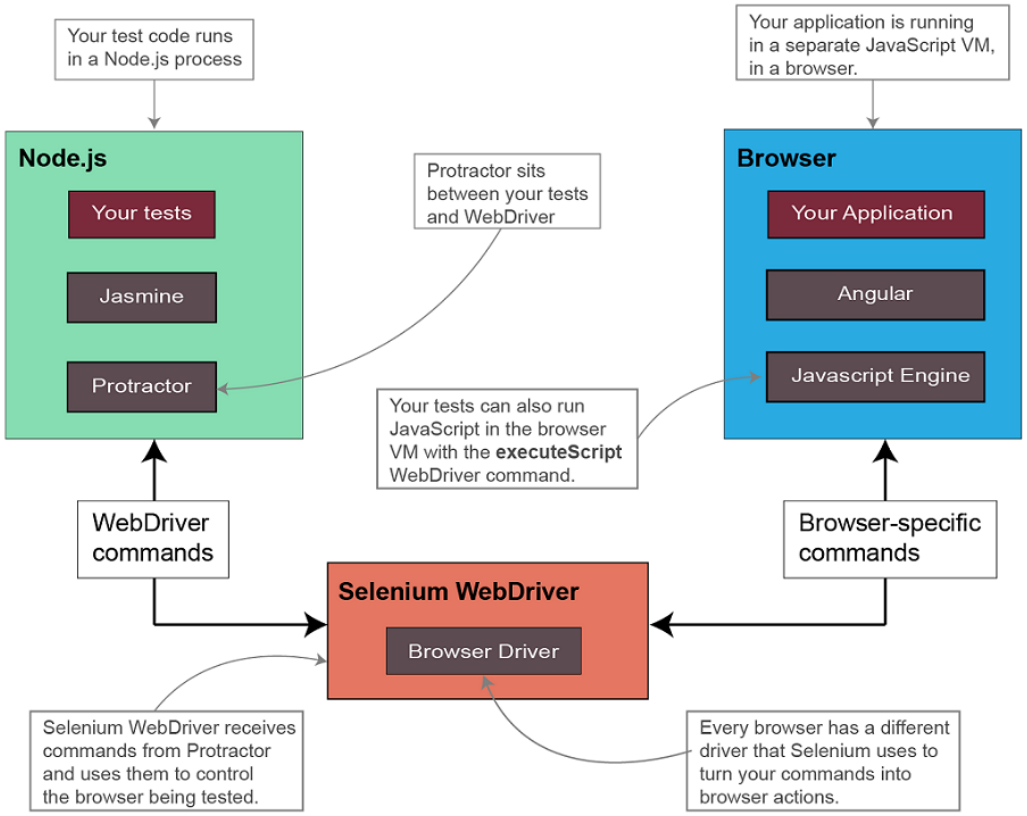

Figure 8.2: Protractor block diagram

Protractor is a Node.js program. After Protractor launches, it hands the execution of your tests over to Jasmine (or Mocha, or whichever test framework you prefer). Protractor wraps the Selenium WebDriver client and adds features that are specific for testing Angular applications. Protractor then communicates to the browser via a Selenium server. The Selenium server controls the browser using vendor-specific browser drivers – there's a different driver for Firefox, Chrome, etc. After the browser driver creates a session, Protractor loads the Angular application under test.

INFO Selenium is an open source project (https://github.com/seleniumHQ) that automates browser testing. The Selenium standalone server handles launching browsers and sending commands to the browser drivers using the WebDriver protocol. The WebDriver protocol is a W3C specification defined at www.w3.org/TR/webdriver. Browser vendors maintain the driver for their browser.

The important thing to take away from all of this is that browser tests run in a browser that is independent of your test. Your tests are running in a Node.js process and the Angular application is running in a browser. Because these are two separate processes, we need to provide some sort of synchronization between the Angular application running in the browser and the Protractor test. Protractor does this by inserting a JavaScript function that runs in the browser and waits for Angular to be stable. In this case, "stable" means that there isn't an event or background task pending that might cause a change to your application's DOM.

8.2 Writing your first Protractor test

In this section, we'll create the files needed to run a simple Protractor test in TypeScript. For your first Protractor test, we'll guide you through some of the features in the contacts list web application. When you run your test, a Chrome browser will launch, navigate to the application's default page, and verify the URL address. The code for this chapter can be found at.

https://github.com/testing-angular-applications/testing-angular-applications/tree/master/chapter08

> **INFO** Protractor documentation can be found at http://protractortest.org. Another good resource is the Protractor cookbook, https://github.com/angular/protractor-cookbook.

8.2.1 File structure

Let's take a look at the files and folder structure from the chapter 8 GitHub repository. These files are the bare minimum files required to run a Protractor test with TypeScript:

```
.
├── e2e/
│   ├── first-test.e2e-spec.ts
│   └── tsconfig.json
├── package.json
└── protractor-first-test.conf.js
```

> **INFO** If you are using the Angular-CLI to create a new Angular project, it generates the scaffold files we need to create our first Protractor test. The chapter 8 GitHub repository folder structure mimics these files.

PACKAGE.JSON

The first file you need to create is the `package.json` shown in listing 8.1. The `devDependencies` specifies that our project uses type definitions specified by the `@types` node modules. These are TypeScript typings for node modules written in JavaScript. In addition to type definitions, we also depend on `typescript`, `ts-node`, and `protractor`.

> **INFO** Protractor is written in TypeScript as of Protractor version 4+ and no additional type definitions are required. Prior to Protractor version 4, the type definitions were found at @types/angular-protractor.

The `scripts` portion of package.json defines a `pree2e` and an `e2e` script. The `pree2e` script launches the `webdriver-manager` node module to download binaries required to control a web browser. When running the `e2e` script to launch Protractor, the `pree2e` script automatically runs first.

Listing 8.1 Node package configuration – package.json

```
{
  "name": "protractor-tests",
  "scripts": {
    "pree2e": "webdriver-manager update --gecko false --standalone false",    (1)
    "e2e": "protractor"                                                        (2)
  },
  "devDependencies": {
    "@types/jasmine": "^2.53.43",
    "@types/jasminewd2": "^2.0.1",
    "@types/selenium-webdriver": "^3.0.0",
    "typescript": "^2.2.1",
    "protractor": "^5.1.1",
    "ts-node": "^2.1.0"
  }
}
```

(1) Automatically downloads support files when running the "e2e" script
(2) Runs the protractor node module when provided with a protractor configuration file

TYPESCRIPT CONFIGURATION FILE

The next file we need is the TypeScript configuration file, `tsconfig.json` in the e2e folder. The configuration file shown in listing 8.2 tells the TypeScript compiler (`tsc` command) which TypeScript files to transpile and which type definitions to use. Also, this file tells the transpiler to emit ES6 (ECMA2015) JavaScript, and that the JavaScript output files should be written to `dist/out-tsc-e2e`.

Listing 8.2 TypeScript compiler configuration – e2e/tsconfig.json

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "declartions": false,
    "emiteDecoratorMetadata": true,
    "experimentalDecorators"
    "lib": [ "es2016" ],
    "module": "commonjs",
    "moduleResolution": "node",
    "outDir": "../dist/out-tsc-e2e",                    (1)
    "sourceMap": true,
    "target": "es6",
    "typeRoots": [
      "../node_modules/@types"                          (2)
    ]
  }
}
```

1. When transpiling TypeScript to JavaScript, write the files to a folder that will be ignored by git
2. Include all type definitions from @types

PROTRACTOR CONFIGURATION FILE

Now that we have the package dependencies and TypeScript support files, we'll need to create the Protractor configuration, `protractor-first-test.conf.js`, shown in listing 8.3. The Protractor configuration file tells Protractor how to launch your test. It can be broken down into several parts: how to launch the browser, the path of the test files, the test framework, and plugins.

Listing 8.3 Protractor configuration file – protractor-first-test.conf.js

```
exports.config = {
  capabilities: {
    browserName: 'chrome'                                    (1)
  },
  directConnect: true,                                       (1)
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  framework: 'jasmine',
  specs: [
    './e2e/first-test.e2e-spec.ts'                           (2)
  ],
  onPrepare: () => {
    require('ts-node').register({
      project: 'e2e'
    });
  }
}
```

1. Specify to launch the Chrome browser directly with ChromeDriver
2. Specifies the TypeScript test files Protractor will launch via ts-node

In this listing `directConnect` and `capabilities` define how to launch and interact with the browser. In `capabilities` we specify we want a Chrome browser. If you don't include any capabilities, Protractor launches Chrome by default. We'll launch the Chrome browser with `directConnect` using the `chromedriver` binary downloaded by `webdriver-manager`.

A few lines further down in test-8-1.conf.js are `framework` and `specs`. We specify that we would like to use Jasmine as the test framework. If we did not specify the `framework`, Protractor would run the test with Jasmine by default. Because you have used the Jasmine test runner in to write unit tests in previous chapters, we'll also use Jasmine for Protractor tests. The test runner will run `e2e/first-test.e2e-spec.ts` listed in the `specs` array.

> **INFO** Protractor allows other test frameworks including Mocha and Cucumber. Protractor provides limited support for these frameworks and is beyond the scope of this book. If you would like to try out other frameworks, see the framework documentation at https://github.com/angular/protractor/blob/master/docs/frameworks.md

The last section of the test-8-1.conf.js is the `onPrepare` method. The `onPrepare` method uses the `ts-node` node module, which lets the Protractor test run the TypeScript files without compilation. We register the e2e directory, which has our tests, with `ts-node`.

PROTRACTOR TEST FILE

The last file you need is the test specification `e2e/first-test.e2e-spec.ts` shown in listing 8.4. Because we use the Jasmine framework, the spec file looks similar to the unit tests we've seen in previous chapters. The first line imports the `browser` from Protractor and uses it to navigate to the contact list web application and validate that the current URL is the `browser.baseUrl`, which is `http://contacts-app-starter.firebaseapp.com`.

Listing 8.4 Test specification – e2e/first-test.e2e-spec.ts

```
import { browser } from 'protractor';

describe('our first protractor test', () => {
  it('should load a page and verify the url', () => {
    browser.get('/');
    expect(browser.getCurrentUrl()).toEqual(browser.baseUrl + '/');
  });
});
```

Now that we have the bare minimum files, we can install the node module dependencies and run our first Protractor test.

8.3 Installing and running

First, we install the node modules defined in the package.json with `npm install`. After the node modules are downloaded, we can launch Protractor using the scripts defined in package.json. We can launch the Protractor test with the command `npm run e2e protractor-first-test.conf.js`. At the beginning of the test, the `pree2e` script downloads the ChromeDriver binary using webdriver-manager. After the files are downloaded, Protractor launches the test using the Protractor configuration file `protractor-first-test.conf.js`.

Protractor starts the new WebDriver instance according to the configuration file. The new WebDriver instance launches a Chrome browser window using the ChromeDriver binary. The console output should look similar to listing 8.5.

Listing 8.5 Running Protractor with the e2e script

```
npm run e2e protractor-first-test.conf.js

> chapter-8-code@0.0.0 pree2e /path/to/protractor-first-test              (1)
> webdriver-manager update --gecko false --standalone false              (2)

[11:48:53] I/file_manager - creating folder /path/to/selenium            (2)
[11:48:54] I/downloader - curl -o /path/to/selenium/chromedriver_2.28.zip
       https://chromedriver.storage.googleapis.com/2.28/chromedriver_mac64.zip
[11:48:54] I/update - chromedriver: unzipping chromedriver_2.28.zip
```

```
      /path/to/selenium/chromedriver_2.28.zip
[11:29:38] I/update - chromedriver: unzipping chromedriver_2.28.zip
[11:29:38] I/update - chromedriver: setting permissions to 0755 for
      /path/to/selenium/chromedriver_2.28
[11:29:38] I/update - chromedriver: chromedriver_2.28 up to date

> chapter-8-starter@0.0.0 e2e /path/to/protractor-first-test            (3)
> protractor "protractor-first-test.conf.js"

[11:29:39] I/launcher - Running 1 instances of WebDriver
[11:29:39] I/direct - Using ChromeDriver directly...                  (4)
Started
.                                                                     (5)

1 spec, 0 failures
Finished in 2.484 seconds
```

1. Before launching the "e2e" test, automatically launch the "pree2e" task
2. Download the Chromedriver binary to the node module webdriver-manager/selenium folder
3. The "e2e" task that we specified in the package.json
4. Launch the Protractor test with "directConnect" to run Chrome
5. The period shows that the first test passes

> **NOTE** Protractor could be launched in a couple of ways. In the first method, if you installed Protractor as a global install, you could run protractor test-8-1.conf.js. The second method is to launch Protractor by referencing the `node_module` folder. From the root directory of chapter 8's sample code, you could run `./node_modules/.bin/protractor protractor-first-test.conf.js` or for Windows machines, `node node_modules/.bin/protractor protractor-first-test.conf.js`. The second method is not recommended and is the equivalent to running the `e2e` script which we have previously covered in the package.json file.

During the test, you'll the Chrome browser launch and close quickly. In this short time, Protractor ran the Jasmine test, launched the Chrome browser, navigated to the contacts list web application, validated the URL, and closed the browser window.

Now that you've run your first Protractor test, we'll expand our tests suite with some new tests. To add tests to the test suite, we'll first need to learn some additional Protractor APIs to handle HTML web element interaction.

8.4 Interacting with elements

In the last section, we saw the bare minimum files needed to write our first Protractor test. In this section, we'll introduce two new Protractor APIs: `element` and `by`. These APIs help you interact with the contact list web application. By the end of this section we'll create several test scenarios around creating a new contact.

The related files from the GitHub repository for the next section are `e2e/add-contact.e2e-spec.ts`, `e2e/add-second-contact.e2e-spec.ts`, and `protractor-add-contact.conf.js`. In `protractor-add-contact.conf.js`, you'll need to copy over most of the test configurations from `protractor-first-test.conf.js`. You also need to change the

Protractor configuration `specs` array from including a single file to using file globbing. This will allow us to test both the `e2e/first-test.e2e-spec.ts` and `e2e/add-contact.e2e-spec.ts` without having to specify the exact files we are using.

Listing 8.6 Protractor configuration file protractor-add-contact.conf.js with file globbing

```
exports.config = {
  capabilities: {
    browserName: 'chrome'
  },
  directConnect: true,
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  framework: 'jasmine',
  specs: [                                                    (1)
    './e2e/add-*contact.e2e-spec.ts'
  ],
  onPrepare: () => {
    require('ts-node').register({
      project: 'e2e'
    });
  }
}
```

(1) Adding specs by file globbing

Since we're using the file globbing option, our Protractor test will run all tests that are in the `e2e` directory that match the file suffix of `*.e2e-spec.ts`.

8.4.1 Test scenario: creating a new contact

Usually its easy to figure out the "happy path" when coming up with end-to-end tests. The happy path is the workflow a user follows to successfully complete a set of tasks. In this case, the happy path is shown in figure 8.3 - the user interacts with the contact list web application to create a new contact. To create a new contact, the user clicks the "+" button, fills out the required fields, and clicks the "Create" button.

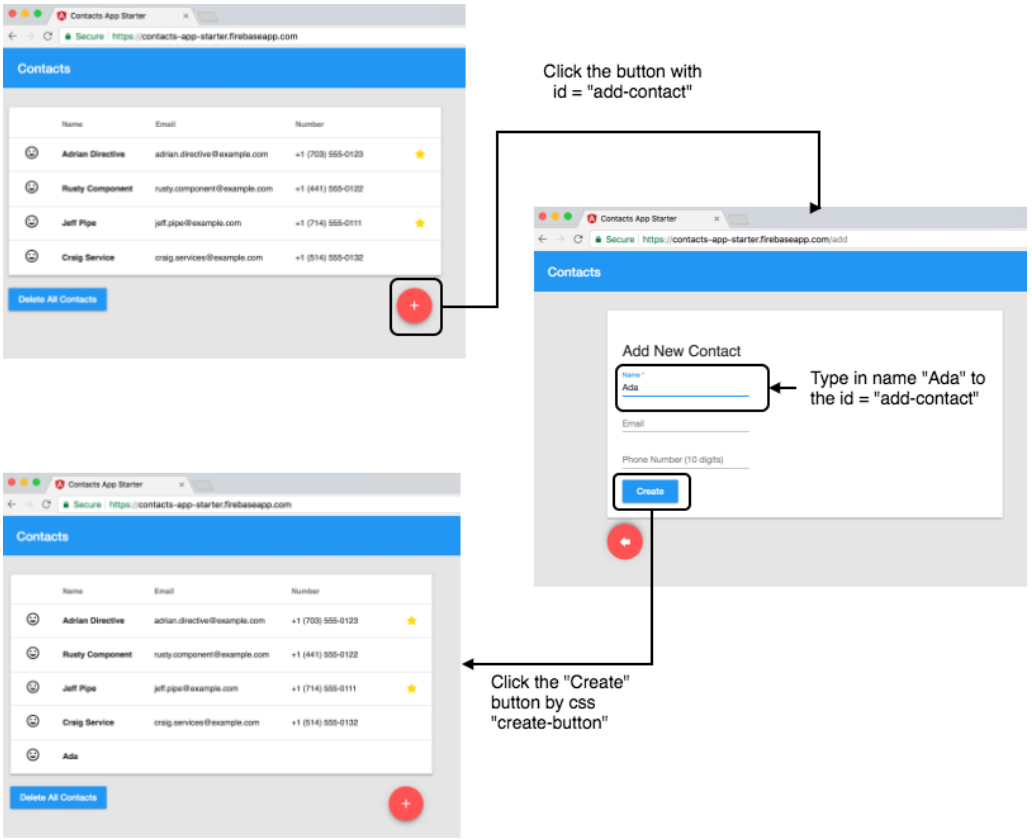

Figure 8.3: Create new contact workflow

To test this workflow, it's helpful to look at the HTML, so that we can identify the elements we need to interact with. The test case will first click on the "+" symbol. In the HTML snippet below, we see the "+" symbol is generated by the mat-icon class "add-fab-icon". We could try to click on the icon, or we could find the web element by either the button or the `<a>` tag. For this example, we'll use the `<a>` tag. Because there could be several `<a>` tags on the webpage, we also need to use the ID `add-contact` to locate the link as shown in the following snippet:

```
<a *ngIf="!isLoading && !deletingContact" id="add-contact" routerLink="/add" mat-fab
       class="add-fab">
  <button mat-fab class="add-fab">
    <mat-icon class="add-fab-icon" mdTooltip="Add new contact">
      add
    </mat-icon>
  </button>
</a>
```

INFO When writing Protractor test, you can use Chrome's developer console to see the HTML for the application. The Chrome developer console is accessed using the keyboard shortcut Cmd+Opt+I on Mac and Ctrl+Shift+I on Windows or Linux.

You can locate and click on the `<a>` tag web element using two Protractor APIs: `by` and `element`. The `by` API has methods to locate web elements using an identifier. The `element` API consumes the object generated by the `by` API and returns a Protractor `ElementFinder` object that represents the web element. After you call the click method on the web element, the contact list web application will navigate to a new page that contains the new contact.

Your test can fill out the form using the ID attribute of the input tag. Similar to the previous step, the HTML input tag can be located using the ID attribute equal to `contact-name`. After locating the element, the test fills out the form with the name "Ada" using the `sendKeys` method. Because this is the first time you are finding a web element, you should verify that the text of the web element matches the expected value. You will need to get the value attribute from the input field and compare it to the text that was just entered. After you assert that the text is Ada, you'll locate the create button by css class `create-button` to add a new contact. After the create contact button is clicked, your test will do one additional check that the route navigated back to the `browser.baseUrl`.

Listing 8.7 Test specification to create a new contact – e2e/add-contact.e2e-spec.ts

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

describe(adding a new contact with only a name', () => {
  beforeAll(() => {
    browser.get('/');
  });

  it('should find the add contact button', () => {
       element(by.id('add-contact')).click();                          (1)
       expect(browser.getCurrentUrl()).toEqual(browser.baseUrl + '/add');
  });

  it('should write a name', () => {
    let contactName = element(by.id('contact-name'));                  (1)
    contactName.sendKeys('Ada');
    expect(contactName).getAttribute('value').toEqual('Ada');
  });

  it('should click the create button', () => {
    element(by.css('.create-button')).click();                         (2)
    expect(browser.getCurrentUrl()).toEqual(browser.baseUrl + '/');
  });
});
```

(1) Find the name input field with the "contact-name" id attribute
(2) Find the create button with the "create-button" css class

When you filled out the name input field on the contact form, you might have noticed that the form also populated an email and phone number field. Another useful test scenario would be to fill out the form completely. In the e2e/add-second-contact.e2e-spec.ts shown in listing 8.8, we moved some of these interactions into the `beforeAll` step because the previous test has already tested loading up the main page and filling out the name field.

Listing 8.8 Test to create another contact – e2e/add-second-contact.e2e-spec.ts

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

describe(adding a new contact with name, email, and phone number', () => {
  beforeAll(() => {
    browser.get('/');
    element(by.id('add-contact')).click();
    element(by.id('contact-name')).sendKeys('Grace');
  });

  it('should type in an email address', () => {
    let email = element(by.id('contact-email'));                    (1)
    email.sendKeys('grace@hopper.com');                             (2)
    expect(email.getAttribute('value')).toEqual('grace@hopper.com');
  });

  it('should type in a phone number', () => {
    let tel = element(by.css('input[type="tel"]'));                 (3)
    tel.sendKeys('1234567890');
    expect(tel.getAttribute('value')).toEqual('1234567890');
  });

  it('should click the create button', () => {
    element(by.css('.create-button')).click();
    expect(browser.getCurrentUrl()).toEqual(browser.baseUrl);
  });
});
```

(1) When declaring the element, webdriver does not search the browser for the web element
(2) When an action is called, the web element is located on the screen and the string is sent to the input tag
(3) Locate the web element by css

NOTE You might have noticed that we are using the returned `element` object and calling the `click` and `getAttribute` methods. The object returned is a Protractor defined object.

```
import {by, element, ElementFinder} from 'protractor';
let email: ElementFinder = element(by.id('contact-email'));
```

Protractor exposes some of the selenium-webdriver APIs via `browser.driver`,. The web element can be accessed by the following code snippet:

```
import {By, WebElement} from 'selenium-webdriver';
let email: WebElement;
email = browser.driver.findElement(By.id('contact-email'));
```

If you can find web elements with both Protractor and selenium-webdriver, why should we use the Protractor APIs to find web elements? The preferred method is to use the Protractor APIs because Protractor will wait for Angular to finish updating the page. You'll learn more about this in the next chapter.

Finally, we can run the Protractor test using the command `npm run e2e protractor-add-contact.conf.js`. When running the `npm` command we can see Protractor launching browsers and creating the contacts we specified in this section.

8.4.2 Test scenario: workflows that do not create a new contact

In the not-so-happy path, a user could enter incorrect data. For instance, the user might forget to enter a required field, enter in a malformed telephone number, or enter in an invalid email address.

In our contact list web application, we'll guide you through test scenarios that fail due to invalid data. For instance, if you try to create a new contact with an invalid email, the result is a modal alert window (figure 8.4).

How do we translate this scenario into a test case? In the GitHub repository this test scenario is covered in `e2e/invalid-contact.e2e-spec.ts` and `protractor-invalid-contact.conf.js`. If you are following along, you'll need to create the file `e2e/test-8-3.e2e-spec.ts`. In this file, you'll write a set of tests that attempt to create a new contact with invalid information. The first enters in a valid name and an invalid email address "baduser.com". After the test clicks on the create button, it should check if the modal alert window is visible and dismiss the message. After dismissing the modal alert message, it's a good idea to verify that the modal alert actually did disappear. To test this we can use another Protractor API called `ExpectedConditions`. The `ExpectedConditions` API combined with `browser.wait` method allows the test to wait for some condition to occur on the web application within a set period of time. In this example we are waiting for the web element to not be present within 5 seconds. Finally, because the test should fail to create a new contact, the test should also check to see if the contact list web application route URL is still on the "/add" route.

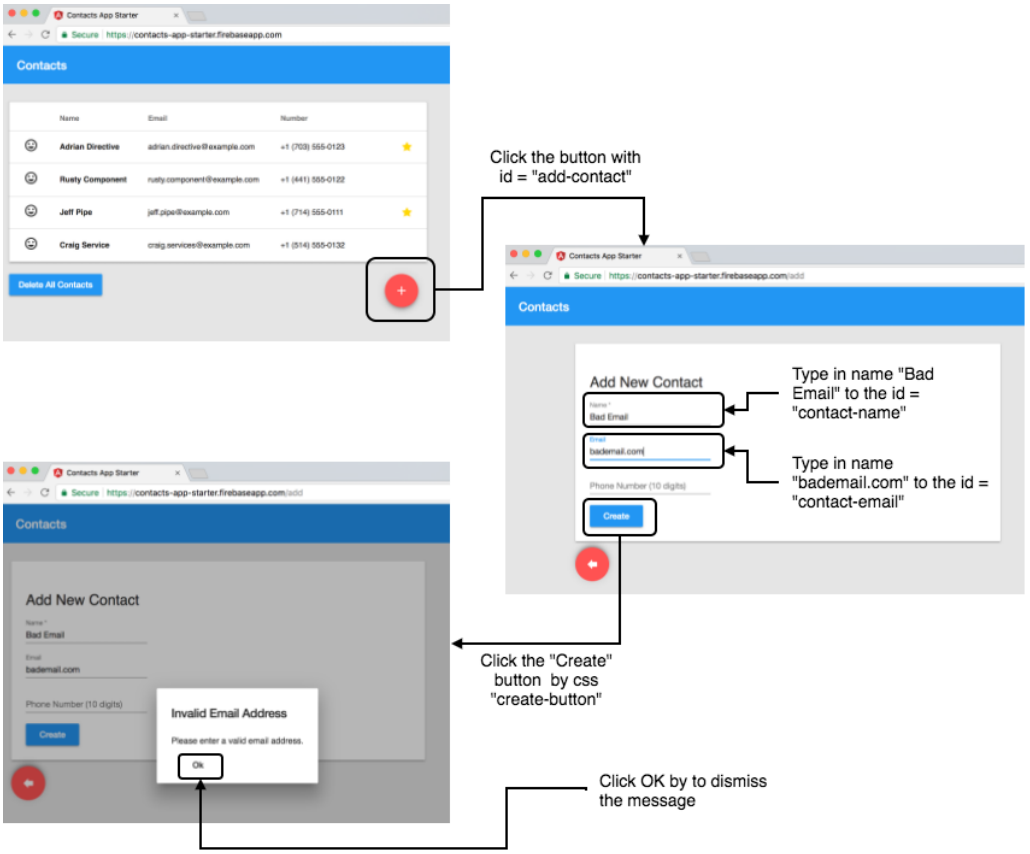

Figure 8.4 Workflow that does not create a new user.

If you try other email inputs, you might also find that you can still create an account if the email field is "@bademail.com". This is obviously incorrect you should either fix the web application or file an issue or bug about this feature.

Now that we have one not-so-happy test scenario, we could try other test scenarios. You could implement more combinations of text inputs; however, this defeats the purpose of having a strong suite of unit tests. This email validation feature should have been covered with a set of text inputs unit tests.

Listing 8.9 Test that does not create a new contact – e2e/invalid-contact.e2e-spec.ts

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

describe('adding a new contact with an invalid email', () => {
  beforeEach(() => {
    browser.get('/add');
```

```
    element(by.id('contact-name')).sendKeys('Bad Email');
  });

  it('should not create a new contact with baduser.com', () => {
    let email = element(by.id('contact-email'));
    email.sendKeys('baduser.com');
    element(by.buttonText('Create')).click();

    let invalidEmailModal = element(by.tagName('app-invalid-email-modal'));
    expect(invalidEmailModal.isPresent()).toBe(true);                    (1)

    let modalButton = invalidEmailModal.element(by.tagName('button'));   (2)
    modalButton.click();

    browser.wait(EC.not(EC.presenceOf(invalidEmailModal)), 5000);        (3)
    expect(invalidEmailModal.isPresent()).toBe(false);                   (4)
    expect(browser.getCurrentUrl()).toEqual(browser.baseUrl + '/add');
  });
});
```

(1) check to see that the modal is present
(2) find the button from the modal web element
(3) waits 5 seconds for the invalid email modal to disappear
(4) check to see that the modal is no longer present

So far, we have written several Protractor tests that have located web elements by either buttonText, id, or css. Unfortunately, not all web elements can be located with just these three locators. In the next section, we'll cover other ways to identify web elements.

8.5 by and element methods

In the previous section, we showed a subset of the different ways to use locators to identify web elements. There are other locators that we did not cover in the previous examples. Table 8.1 lists the common locator methods and where to use them.

Table 8.1: Locating web elements with the by API

Locator	Usage

`by.css`	**Find a web element by css**
	HTML:
	`<input class="contact-email" id="contact-email" type="email">`
	Protractor:
	`let e1 = element(by.css('.contact-email'));`
	`let e2 = element(by.css('#contact-email'));`
	`let e3 = element(by.css('input[type="email"]'));`

by.id	**Find a web element by id.**
	HTML:
	`<input class="contact-email" id="contact-email" type="email">`
	Protractor:
	`let email = element(by.id('contact-email'));`

by.buttonText
by.partialButtonText
	Find a button with the matching text.
	HTML:
	`<button>Submit Contact</button>`
	Protractor:
	`let fullMatch = element(by.buttonText('Submit Contact'));`
	`let partialMatch = element(by.partialButtonText('Submit'));`

by.linkText
by.partialLinkText
	Find a link by matching text.
	HTML:
	`<a href="/add">Add contact</a>`
	Protractor:
	`let fullMatch = element(by.linkText('Add contact'));`
	`let partialMatch = element(by.partialLinkText('contact'));`

by.tagName	**Find a web element by tag name.**
	HTML:
	`<app-contact-detail>…</app-contact-detail>`
	Protractor:
	`let tag = element(by.tagName('app-contact-detail'));`

by.xpath	**Find a web element by xpath. Using xpath as a locator strategy can create brittle tests requiring high maintenance. We recommend to not use xpath as a locator strategy.**
	HTML:
	`<ul><li><a>Foobar</a></li></ul>`
	Protractor:
	`let xpath = element(by.xpath('//ul/li/a'));`

by.binding	**Find a web element by binding for objects in AngularJS. Currently this is not implemented for Angular.**
	HTML:
	`<span ng-bind="contact.name"></span>`
	Protractor:
	`let binding = element(by.binding('contact.name'));`

`by.model`	**Find a web element by model in AngularJS. Currently this is not implemented for Angular.**
	HTML:
	`<input ng-model="contact.name">`
	Protractor:
	`let model = element(by.model('contact.name'));`

NOTE **Protractor's By API is not the same as a selenium-webdriver's By API**

We should emphasize that Protractor's By and selenium-webdriver's By are different. This is important because Protractor exposes some of the selenium-webdriver APIs and remembering the difference will hopefully help you during debugging. The following code snippet shows how to use the Protractor `by` API.

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

element(by.buttonText('Create'));
```

Since we're using the Protractor APIs here, we should use the Protractor By API. When we use selenium-webdriver APIs that are exposed through Protractor to find a web element, we should consistently also use selenium-webdriver By locator. The following code snipped shows how to use the selenium-webdriver `by` API.

```
import {browser} from 'protractor';
import {By as WebdriverBy} from 'selenium-webdriver';

browser.driver.findElement(WebdriverBy.css('.contact-email'));
```

The following code snippet is subtlety different from previous examples. It doesn't work because we're passing the selenium-webdriver `findElement` method a Protractor `By` finder.

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

browser.driver.findElement(by.buttonText('Create'));
```

So far, we've covered `sendKeys`, `click`, and `getAttribute`; however, there are other ways to interact with web elements. Table 8.2 covers the commonly used `element` methods. All of these methods return a WebDriver promise. Protractor's test framework takes those WebDriver promises and makes these browser interactions appear synchronous. Synchronizing these asynchronous WebDriver calls removes some of the complexity that these WebDriver promises introduce.

Table 8.2: Interacting with web elements with the element API

Element method	Usage

`getWebElement`	Occasionally you'll need to access selenium-webdriver `WebElement`'s APIs that are not available from Protractor's element object. One example of this would be verifying a web element's x and y location via the `getLocation` method, which exists only on the `WebElement` object. After calling the `getWebElement` method, you will need to wait for the WebDriver promise for the `WebElement` to resolve.
	
	`let button = element(by.css('.contact-email'))`
	`  .getWebElement();`
	`button.getLocation().then(point => {`
	`  console.log('x = ' + point.x + ', y = ' + point.y);`
	`});`

`isPresent`
`isElementPresent`
	When testing Angular structural directives like `*ngIf`, you need to call `isPresent` to validate if a web element exists on the screen. In the contact list web application, after the test enters the name of the new contact, the email field appears and could be tested like so:
	
	`browser.get('/add');`
	`expect(element(by.id('contact-email'))`
	`  .isPresent()).toBe(false);`
	`element(by.id('contact-name')).sendKeys('foo');`
	`expect(element(by.id('contact-email'))`
	`  .isPresent()).toBe(true);`

`getTagName`	When writing tests, you can use identifiers like css to find the web element. Use the `getTagName` method to validate the current tag that is being returned from Protractor.
	
	`browser.get('/add');`
	`let body = element(by.tagName('body'));`
	`let mdToolbar =`
	`  body.element(by.css('[color="primary"]'));`
	`expect(mdToolbar.getTagName()).toBe('mat-toolbar');`

getCssValue	Use to get the value of a given css property.
	
	`browser.get('/add');`
	`let toolbar = element(by.tagName('mat-toolbar'));`
	`expect(toolbar.getCssValue('background-color')).toBe('rgba(33, 150, 243, 1)');`

getAttribute	When typing in values into input fields, you can validate the input field was entered properly using the `getAttribute` method. To get the contents of an input field, you will need to get the 'value' attribute. A common mistake is to try to use the `getText` method to get the text from an input field.
	
	`browser.get('/add');`
	`let email = element(by.id('contact-email'));`
	`email.sendKeys('foobar');`
	`expect(email.getAttribute('value')).toBe('foobar');`

getText	When you use the `getText` method, Protractor will return a promise for the text that appears on the web application from that web element. Note that to get the text for an input field, you'll need to use `getAttribute('value')` as above.
	
	`browser.get('/');`
	`element(by.tagName('tbody')).getText().then(text => {`
	`  console.log(text);`
	`  expect(text.match(/craig.service@example.com/)`
	`  .index > 0).toBe(true);`
	`  expect(text.match(/something that does not match/))`
	`  .toBe(null);`
	`});`

sendKeys	Use to simulate typing in text – for example, to fill out an input field.
	`browser.get('/add');`
	`element(by.id('contact-name')).sendKeys('foobar');`
	`expect(element(by.id('contact-name'))`
	`  .getAttribute('value')).toBe('foobar');`

`clear`	**Clears text from an input field.**
	`browser.get('/add');`
	`let name = element(by.id('contact-name'));`
	`name.sendKeys('foo bar');`
	`name.clear();`
	`expect(name.getAttribute('value')).toBe('');`

`isDisplayed`	**You can check if an element is present in the but hidden from view with `isDisplayed`. If there are web elements that are hidden but still part of the DOM, Protractor will return that it is present but not displayed.**
	
	`browser.get('/add');`
	`let contactName = element(by.id('contact-name'));`
	`expect(contactName.isDisplayed()).toBe(true);`
	`// Change the input to not be visible by style.`
	`browser.executeScript("arguments[0].setAttribute('style',`
	`'display:none;')", contactName.getWebElement());`
	`expect(contactName.isPresent()).toBe(true);`
	`expect(contactName.isDisplayed()).toBe(false);`

In this section, we reviewed commonly used methods to interact with web elements. In the next section we'll examine how to work with a collection of web elements.

8.6 Interacting with a list of elements

Interacting with a list of elements is similar to interacting with a single element. Finding web elements is asynchronous, whether it's a single element or a collection, so the result is a promise. A common gotcha is to try to iterate over the collection of web elements with a for-loop. You can't loop through a promise, so instead we'll use the Protractor API methods for `element.all`. For the contacts list web application, you can call `element(by.tagName('tbody')).element.all(by.tagName('tr'))` to get the array of web elements. In the following sections, we'll cover several methods that will help you.

8.6.1 Filtering web elements

Let's consider creating a new contact in the contact list web. How do we validate that the new contact exists in the list of contacts? We could find the `tbody` web element, get all the text from the table body, and create a regular expression to match the new contact. What if there are two contacts with the same phone number? How can we get the information for a single

contact? We use the filter function, which will find a subset of contacts from the contact list. In figure 8.5, we show how to find a contact that matches the name that is equal to Craig Service.

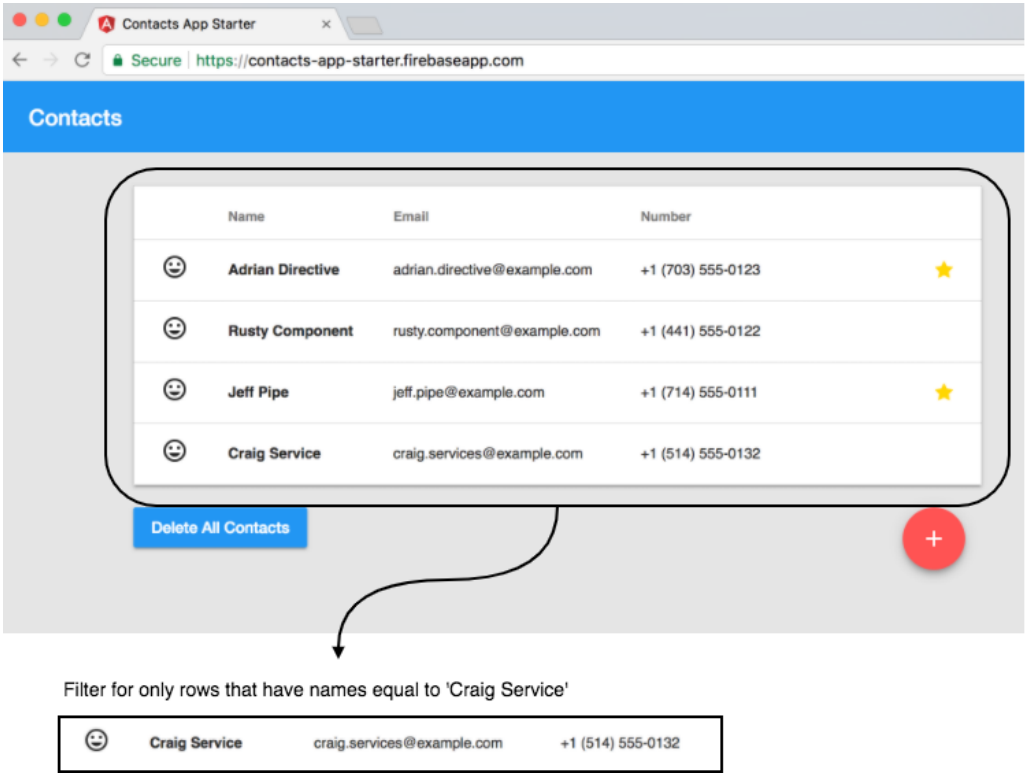

Figure 8.5 Filtering strategy for names equal to 'Craig Service'

When we call the filter function in figure 8.6, the returned object is an array of web elements that satisfy the callback function. In this example, the filter callback function, `filterFn`, returns true when the name matches Craig Service. See the diagram below for the function signature:

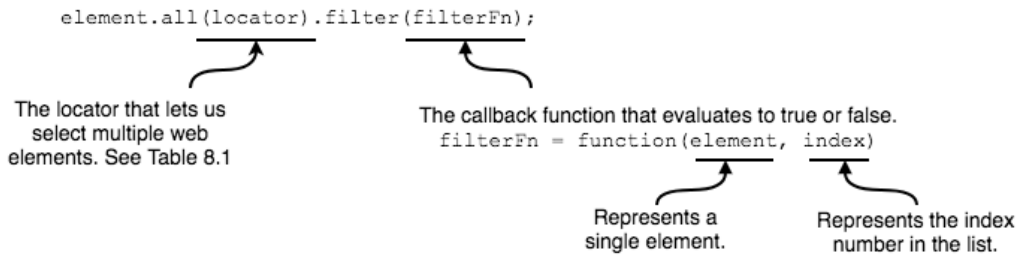

Figure 8.6 Filter function

For this example, look at the contact component template shown in listing 8.10. You might notice that you can't call `element.all(by.tagName('tr'))`, because that would also include the table headers. You might wonder if we could use the `*ngFor` to identify just the rows of the contacts. That doesn't work, since `*ngFor` tells Angular how to modify the DOM's structure, and that attribute isn't included in the rendered output.

Listing 8.10 HTML of the contact list – src/app/contacts/contact-list/contact-list.component.html

```
<table class="mdl-data-table mdl-js-data-table mdl-shadow--2dp">
  <thead>
    <tr>
      <th class="mdl-data-table__cell--non-numeric"></th>
      <th class="mdl-data-table__cell--non-numeric">Name</th>
      <th class="mdl-data-table__cell--non-numeric">Email</th>
      <th class="mdl-data-table__cell--non-numeric">Number</th>
      <th class="mdl-data-table__cell--non-numeric"></th>
      <th class="mdl-data-table__cell--non-numeric"></th>
      <th class="mdl-data-table__cell--non-numeric"></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let contact of contacts" (click)="onSelect(contact)">    ❶
      <td class="mdl-data-table__cell--non-numeric"
          (click)="onClick(contact)">
        <mat-icon>mood</mat-icon>
      </td>
      <td class="mdl-data-table__cell--non-numeric"
          (click)="onClick(contact)">                                    ❷
        <strong>{{ contact.name }}</strong>
      </td>
      <td class="mdl-data-table__cell--non-numeric"
          (click)="onClick(contact)">{{ contact.email }}</td>
      <td class="mdl-data-table__cell--non-numeric"
          (click)="onClick(contact)">
        {{ contact.number | phoneNumber : "default" : contact.country : true }}
      </td>
    </tr>
  </tbody>
</table>
```

1. Find the table rows that represent contacts
2. Use the table column to check the name

Listing 8.11 shows a Protractor test against the above HTML that uses the filter method. First, we find the `tbody` web element using the `by.tagName` locator. Within that `tbody` we then get all the table rows and assign them to `trs`. Next, we filter the table rows in `trs` to find the one we want. We pass `filter()` a callback function that evaluates to if the text in the row matches "Craig Service". The resulting list only includes web elements for which the callback function returned `true`.

Listing 8.11 Filter for a contact – e2e/contact-list.e2e-spec.ts

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

describe('the contact list', () => {
  it('with filter: should find existing contact "Craig Service"', () => {
    let tbody = element(by.tagName('tbody'));
    let trs = tbody.all(by.tagName('tr'));                          (1)
    let craigService = trs.filter(elem => {
      return elem.all(by.tagName('td')).get(1).getText()            (2)
        .then(text => {
          return text === 'Craig Service';                          (3)
      });
    });
    expect(craigService.count()).toBeGreaterThan(0);                (4)
    expect(craigService.all(by.tagName('td')).get(2).getText())     (5)
      .toBe('craig.services@example.com');
  });
});
```

1. Find the array of table rows that represent contacts that exist within the table body
2. Use the second table column to compare the contact name.
3. getText returns a promise of the Boolean evaluation where text === 'Craig Service'
4. Check to see if craigService exists
5. As an additional check, verify that the third column is the correct email address

To verify that we found the correct row for "Craig Service", we could also check that we found only one element and that the email matches "craig.services@example.com".

8.6.2 Mapping the contact list to an array

Let's consider a different scenario where you need to test all the contacts on the contact list. Instead of writing a filter function for each contact, you could use the `map` function. The map function converts the web elements returned from the `element.all` to an array shown in figure 8.7.

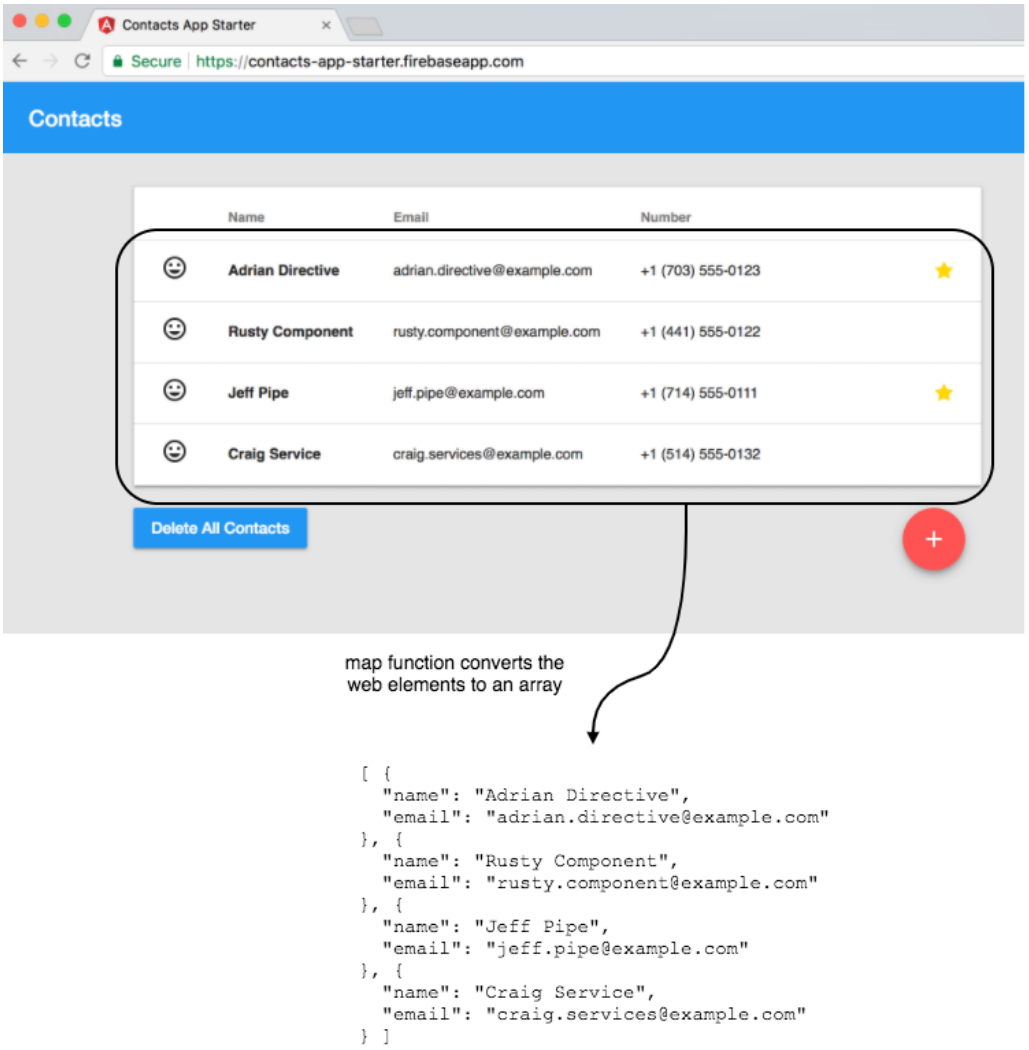

Figure 8.7 Convert the web elements to an array using the map function

Before we use the map function (in figure 8.8) we should review the `map` function for `element.all` shown in the diagram below:

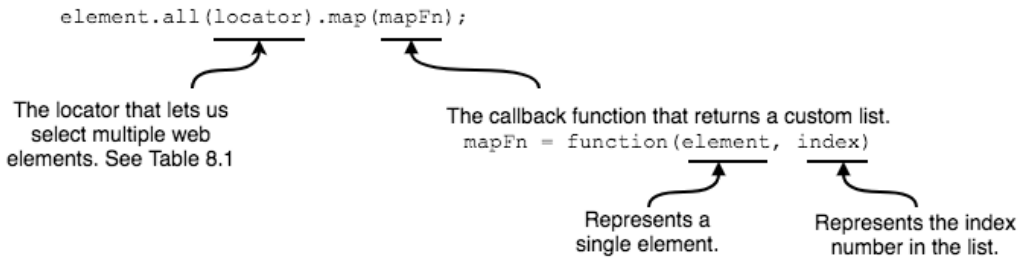

Figure 8.8 Map function

In listing 8.12, we use `map` to transform the contact list into an array of objects that implement the `Contact` interface. To validate that the contact list appears as expected, we create an expected list of contacts using the same `Contact` interface and compare them at the end of the test.

Listing 8.12 Check all the contacts appear as expected with map – e2e/contact-list.e2e-spec.ts

```
import { browser, by, element } from 'protractor';
import { promise as wdpromise } from 'selenium-webdriver';

export interface Contact {
  name?: string;
  email?: string;
  tel?: string;
}

describe('the contact list', () => {
  let expectedContactList: Contact[] = [{
    name: 'Adrian Directive',
    email: 'adrian.directive@example.com',
    tel: '+1 (703) 555-0123'
  }, {
    name: 'Rusty Component',
    email: 'rusty.component@example.com',
    tel: '+1 (441) 555-0122'
  }, {
    name: 'Jeff Pipe',
    email: 'jeff.pipe@example.com',
    tel: '+1 (714) 555-0111'
  }, {
    name: 'Craig Service',
    email: 'craig.services@example.com',
    tel: '+1 (514) 555-0132'
  }];

  beforeAll(() => {
    browser.get('/');
  });

  it('with map: should create a map object', () => {
```

```
    let tbody = element(by.tagName('tbody'));
    let trs = tbody.all(by.tagName('tr'));
    let contactList = trs.map(elem => {                                    (1)
      let contact: Contact = {};
      let promises: any[] = [];
      let tds = element.all(by.tagName('td'));                             (2)
      promises.push(tds.get(0).getText().then(text => {                    (3)
        contact.name = text;
      }));
      promises.push(tds.get(1).getText().then(text => {                    (3)
        contact.email = text;
      }));
      promises.push(tds.get(2).getText().then(text => {                    (3)
        contact.tel = text;
      }));

      return Promise.all(promises).then(() => {                            (4)
        return contact;
      });
    });
    expect(contactList).toBeDefined();                                     (5)
    contactList.then((contacts: Contact[]) => {                            (6)
      expect(contacts.length).toEqual(4);
      expect(contacts).toEqual(expectedContactList);                       (7)
    });
  });
});
```

(1) For each contact, run the mapFn callback for each tr element
(2) Get the table row columns
(3) Get the text and set the value to the corresponding contact property then push the promise to the promises array
(4) Resolve the promises to set the properties to a single contact and return the contact
(5) Check that the contact list is not undefined
(6) Cast the resolved contact list to a Contact array
(7) Check that the contacts list is equal to the expected contacts list

In listing 8.12, we use a promise array to keep track of promises to set the name, email, and telephone number to the contact when calling `getText`. After the promises are created and added to the promise array, we call `Promise.all`. When we call `then` on the `Promise.all`, all the promises in the array are resolved. In this case, the contact properties are set. Finally, we return the contact for that row.

The map function iterates through all the web elements and returns a promise that resolves to the contact list. Next, the test calls `then` to get the contact list array. With the contact list array, the test can now verify if the expected contact list matches the one from the web application.

8.6.3 Reduce

Another possible test scenario might be testing that just the names match. As before, we could use map to create an array of names. However, an alternative solution would be to use

the reduce function, which can turn a collection of contact web elements into a single string of names (figure 8.9).

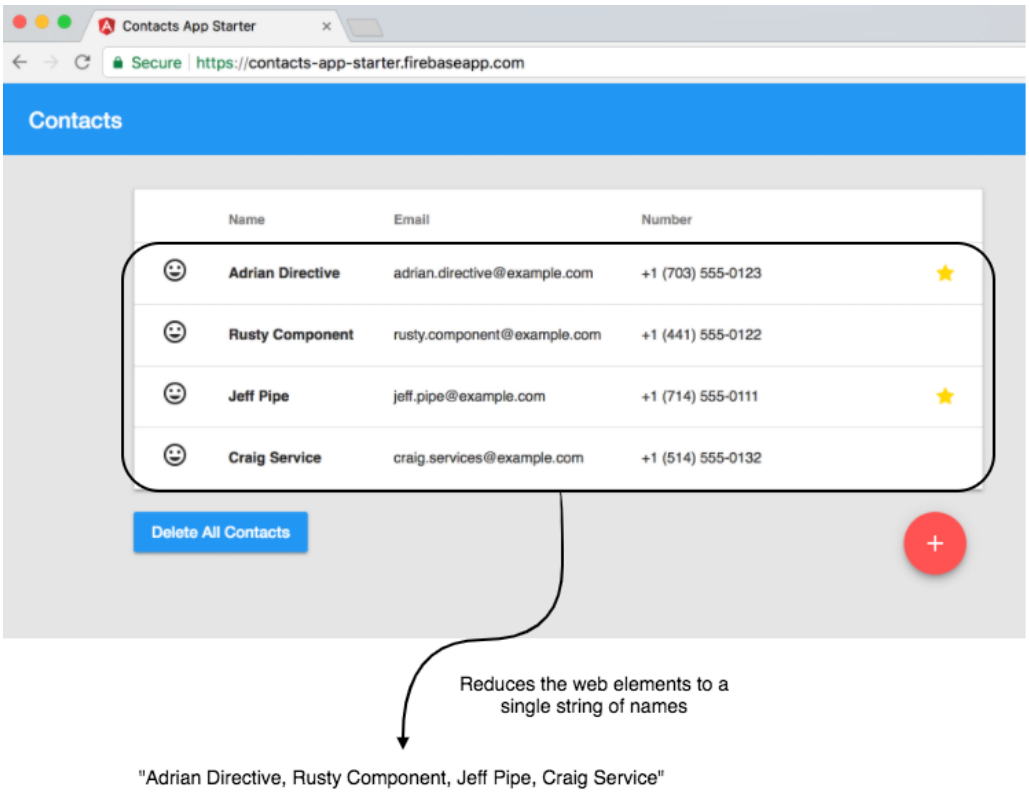

Figure 8.9 Reduce the contact list to a single string of names

The reduce function applies a callback to each element of the array and accumulates the result in a single value. The method signature is shown below in figure 8.10.

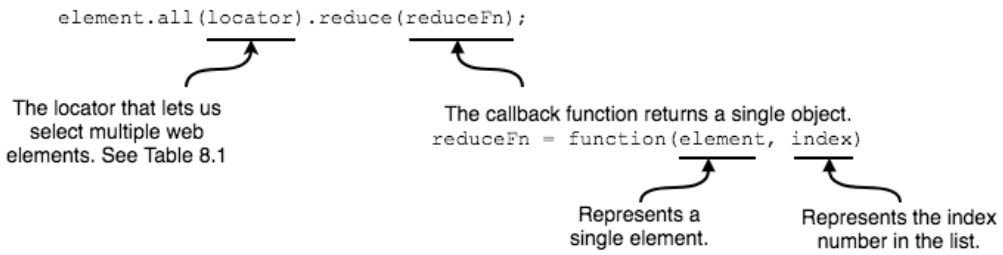

Figure 8.10 Reduce function

In our contacts web application, we gather the contact names into a single string to check the default values. The reduce function returns a comma-delimited list of names (listing 8.13).

Listing 8.13 Reduce the list of elements to a single string – e2e/contact-list.e2e-spec.ts

```
describe(the contact list', () => {
  beforeAll(() => {
    browser.get('/');
  });

  it('with reduce: get a list of contact names', () => {
    let tbody = element(by.tagName('tbody'));
    let trs = tbody.all(by.tagName('tr'));                          (1)
    let contacts = trs.reduce((acc, curr) => {                      (2)
      let name = curr.all(by.tagName('td')).get(0);                 (3)
      return name.getText().then(text => {
        return acc + ', ' + text;                                   (4)
      });
    });
    expect(contactList)
        .toEqual('Adrian Directive, Rusty Component, Jeff Pipe, ' +
        'Craig Service');
  });
});
```

(1) Get each table rows from the table body
(2) Call reduce to get a string of contacts
(3) Get the name column
(4) Accumulate the names as a comma-delimited string

The `curr` parameter in the callback represents the table row web element. The callback gets the text from the first column. Then, we take the text and concatenate it into the accumulator. Finally, the test checks if the accumulated string matches the expected list of names.

8.7 Page objects

Let's say the developer (whether you or someone else) changes the ID from "add-contact" to "create-contact". After the changes are published, if your tests were not updated, they would

fail because the ID is no longer `add-contact`. Having hard coded strings like IDs or class names makes tests harder to maintain – it can lead to you having to manually update all your tests after a single, simple code change.

One way to make tests more maintainable is to use a common design pattern called *page objects.* Page objects organize your test code around logical interactions with your web app, instead of with raw elements. Instead of finding the create button then calling the click method, a page object would instead have a `clickCreateButton`. If the locator for the create button needs to change, you can fix it in a single location.

Previously you wrote the "create contact" test that typed in the new contact name and clicked the create button. The examples so far have had IDs and class names that provide helpful hints as to their function. In real world applications, these identifiers are not always so helpful and can be arbitrary.

Listings 8.14 and 8.15 consolidate all the interactions for creating a new contact in a `ContactListPageObject` and a `NewContactPageObject`. The page objects group the webdriver commands into typical interactions. Each constructor sets the element finders for the page view; however, Protractor doesn't actually find the element until there's a call that interacts with it, like a `click()` or `sendKeys()`.

Listing 8.14 Contact List – e2e/po/contact-list.po.ts

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

export class ContactListPageObject {
  plusButton: ElementFinder;

  constructor() {
    this.plusButton = element(by.id('add-contact'));                ❶
  }

  clickPlusButton() {
    this.plusButton.click();
    return new NewContactPageObject();                              ❷
  }

  navigateTo() {
    browser.get('/');
  }
}
```

❶ Create reusable element finder objects
❷ When clicking on the plus button, since the page navigates from the contact-list, call the constructor to return a page object for creating a new contact

Listing 8.15 Contact List – e2e/po/new-contact.po.ts

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

export class NewContactPageObject {
  inputName: ElementFinder;
  inputEmail: ElementFinder;
```

```
  inputPhone: ElementFinder;

  constructor() {
    this.inputName = element(by.id('contact-name'));
    this.inputEmail = element(by.id('contact-email'));
    this.inputPhone = element(by.css('input[type="tel"]'));
  }

  setContactInfo(name: string, email: string, phoneNumber: string) {
    this.inputName.sendKeys(name);                                    (1)
    if (email) {                                                      (2)
      this.inputEmail.sendKeys(email);
    }
    if (phoneNumber) {                                                (2)
      this.inputPhone.sendKeys(phoneNumber);
    }
  }

  clickCreateButton() {
    this.element(by.buttonText('Create')).click();
           return new ContactListPageObject();
  }

  getName() {                                                         (3)
    return this.inputName.getAttribute('value');
  }
}
```

(1) **Find the element finder object then send the keys**
(2) **Send keys for optional fields**
(3) **Implicitly return a webdriver promise for a string**

Now that we've seen how to make page objects, we can refactor our create contact test in listing 8.16. First, the test creates the `contactList` object and navigates to the contact list page. The next `it()` block clicks the plus button and verifies that the current URL is the create contact page. On the create contact page, the test fills out the name input field and email input field. After the fields are filled, the test verifies that the input values match. Finally, the test clicks the create button and returns to the contact list page.

Listing 8.16 Refactor creating a new contact – e2e/page-object.e2e-spec.ts

```
import { ContactListPageObject } from './po/contact-list.po.ts';
import { NewContactPageObject } from './po/new-contact.po.ts', , Contact }

describe('contact list', () => {
  let contactList: ContactListPageObject;
  let newContact: NewContactPageObject;

  beforeAll(() => {
    contactList = new ContactListPageObject();
  });

  describe('add a new contact', () => {
    beforeAll(() => {
      contactList.navigateTo();
```

```
  });

  it('should click the + button', () => {
    newContact = contactList.clickPlusButton();
    expect(newContact.getCurrentUrl()).toBe(browser.baseUrl + '/add');
  });

  it('should fill out form for a new contact', () => {
    newContact
      .setContactInfo('Mr. Newton', 'mr.newton@example.com', null);
    expect(newContact.getName()).toBe('Mr. Newton');
    expect(newContact.getEmail()).toBe('mr.newton@example.com');
    expect(newContact.getPhone()).toBe('');
  });

  it('should click the create button', () => {
    contactList = newContact.clickCreateButton();
    expect(contactList.getCurrentUrl()).toBe(browser.baseUrl + '/');          (1)
  });
});
```

(1) Instead of using browser.getCurrentUrl(), using contactList.getCurrentUrl provides hints to what the URL should be

Instead of importing in Protractor's `browser`, `by`, and `element`, the test imports the page objects and uses the only the methods from the page objects for navigation and validation.

8.8 Summary

This chapter introduced testing as a user with an end-to-end testing framework. Although it may be tempting to create an end-to-end test for every possible use case, it's better to do the bulk of your testing with unit tests, which run faster and are easier to debug. If you need to create end-to-end tests, you should use the Angular supported end-to-end test framework, Protractor. Some of the Protractor topics we learned include the following:

- **Protractor file structure:** The Angular CLI provides a scaffold of Protractor files that can be used as a good starting point. The setup allows us to use the built in TypeScript support when writing our Protractor tests.
- **Finding elements:** Protractor has many ways to locate objects on the screen. If there is more than one web element, Protractor can find an array of elements.
- **Page objects:** Use page objects when writing Protractor tests. It allows for locators to be initialized in a single location and allows us to encapsulate actions into methods.

Now that we've reviewed the Protractor basics, the next two chapters will cover more advanced Protractor topics. In chapter 9 we'll learn how Protractor waits for an Angular application to be stable, how to create your own wait methods, and when to use them. In chapter 10 we'll cover advance Protractor configuration settings, using Protractor in continuous integration, and taking screenshots during tests.

9

Understanding timeouts

This chapter covers

- **Understanding and avoiding the causes of timeout errors in Protractor**
- **Waiting for specific changes in your app, rather than relying on** `browser.sleep()`
- **Understanding flakiness and eliminating it with Protractor**

Now that you know how to make basic end to end tests for Angular apps, let's talk about one of the most frequent issues you might run into. Timeout errors are the most common problem people encounter when using Protractor for the first time. Understanding what causes them and how to fix them requires a clear understanding of how browser tests run, as well as how Protractor makes your tests more reliable by waiting for Angular to be stable while running a test.

In this chapter, we'll demonstrate how to avoid the common timeout-related pitfalls that new Protractor users stumble into. On the way, we'll learn how Angular's change detection works and how Protractor integrates with it. We'll also learn some advanced techniques for making your own waiting logic. It may seem like a lot to cover, but it makes sense when you see how everything fits together. The example code from this chapter can be found at `https://github.com/testing-angular-applications/testing-angular-applications/tree/master/chapter09`.

9.1 Kinds of timeouts

Because Protractor tests involve many different pieces working together, there are different kinds of timeouts. For example, Jasmine will mark your test as failed if it takes too long to complete. WebDriver will throw an error if a browser command takes too long. For this

chapter, we're only concerned about one kind timeout – the timeout that occurs if Protractor waits too long for Angular to be stable.

What is 'flakiness?'

According to Dictionary.com, 'flaky' is slang for 'eccentric' or 'crazy'. When we say a test is 'flaky', what we mean is that it's non-deterministic – it might fail even though there's nothing wrong with our app. We want to avoid flakes – if the tests can fail when nothing has changed in the app then people tend to ignore them and they become less useful.

One potential cause of flakiness is having a test read the DOM of a page while Angular is in the middle of updating it. We could avoid this by adding sleep commands after every step in our tests that might cause Angular to update the page, but this would slow down our test runs, and it's not guaranteed to always work. This is why Protractor synchronizes your tests with Angular – to prevent flaky test failures.

Waiting for Angular to be stable prevents your test from interacting with the page while Angular is in the middle of an update, thus making your tests less flaky. However, sometimes it can cause problems, particularly when you need to test a page that isn't part of an Angular app.

9.2 Testing pages without Angular

Remember from chapter 8 that Protractor intercepts the commands your test sends to WebDriver and automatically waits for Angular to be ready. This mechanism is one of the biggest advantages of using Protractor, but sometimes it gets in the way.

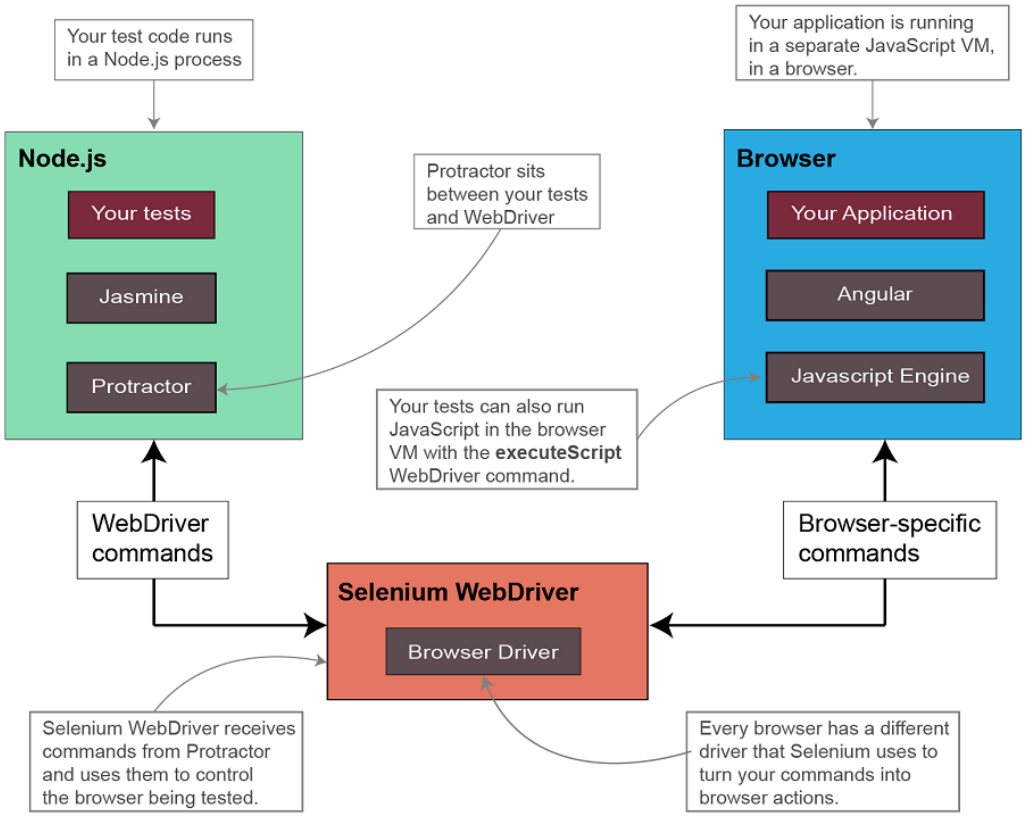

Figure 9.1: How Protractor interacts with WebDriver

One of the first problems new users of Protractor run into is writing a test that logs into their application. If the authentication page is just static HTML and not part of the Angular app, then Protractor will throw an 'Angular could not be found on the page' error. This is because Protractor is expecting to see a page that's part of an Angular app and can't tell the difference between a page that's not supposed to have Angular and one that's just broken.

Our example contacts app doesn't have authentication, but let's pretend for a minute that it does and that we need to log in before we can run our tests. We've added a fake login page at `/assets/login.html`. This is just a simple HTML file that's bundled with the contacts app, but it doesn't actually do anything.

9.2.1 Disabling waitForAngular

Let's make a test that navigates to the login page using `browser.get('/assets/login.html')`. Running the test produces this error:

```
1) the contact list should find the title
- Failed: Angular could not be found on the page http://localhost:4200/assets/login.html. If
      this is not an Angular application, you may need to turn off waiting for Angular.
```

What does the error mean? As we saw earlier, Protractor automatically intercepts the commands your tests send to WebDriver and inserts commands that communicate with Angular and wait for your application to be ready for testing. When you navigate to a page that isn't an Angular app, Protractor throws an error because it can't find Angular. To fix the error, we simply tell Protractor not to wait for Angular:

```
it('should be able to log in', () => {
  browser.waitForAngularEnabled(false);
  browser.get('/assets/login.html');
  element(by.css('input.user')).sendKeys('username');
     element(by.css('input.password')).sendKeys('password');
  element(by.id('login')).click();
);
```

Now, suppose that we want to test that clicking the login button actually redirects to the contact list page. We might add the following to our test:

```
  it('should be able to login', () => {
    browser.waitForAngularEnabled(false);
    browser.get('/assets/login.html');
    element(by.css('input.user')).sendKeys('username');
    element(by.css('input.password')).sendKeys('password');
    element(by.id('login')).click();

    const list = element(by.css('app-contact-list tr'));
    expect(list.getText()).toContain('Jeff Pipe');
  });
```

We're almost there, but now the test is failing for a different reason. We've turned off waiting for Angular, which means Protractor now has no way to know what the application is doing.

9.2.2 Automatically waiting for Angular

Here's the error we see when we run the test from the previous section:

```
1) the contact list should find the title
  - Failed: No element found using locator: By(css selector, app-contact-list tr)
```

The problem is that when we're looking for the contact list, our app is still loading. Because we've told Protractor not to wait for Angular, it goes right ahead and looks for the contact list, then fails when it doesn't find it. The fix for this is simple – we tell Protractor to start waiting for Angular again after we click the "login" button. Here's the full, working test:

Listing 9.1 Testing a login page

```
it('should be able to login', () => {
```

```
  browser.waitForAngularEnabled(false);       ❶
  browser.get('/assets/login.html');          ❷
  element(by.css('input.user')).sendKeys('username');
  element(by.css('input.password')).sendKeys('password');
  element(by.id('login')).click();

  browser.waitForAngularEnabled(true);        ❸
  const list = element(by.css('app-contact-list tr'));
  expect(list.getText()).toContain('Jeff Pipe');
});
```

❶ Disable automatically waiting for Angular
❷ Test the login page
❸ Re-enable waiting so we can test the application

Now our test will not wait for Angular on the login page, but will go back to waiting for Angular when we return to the app. Why do we need to explicitly disable waiting? Why couldn't Protractor just detect if there's an Angular app on the page and skip waiting if it doesn't find one? Protractor has no way of telling the difference between a page that isn't an Angular app and an app that's just loading very slowly, so we need to let Protractor know that we're intentionally sending it to a non-Angular page. Being explicit about our intentions prevents hard to debug issues with our tests, especially when we're relying on Protractor to automatically wait for Angular to finish updating the page. It's important that we know right away if that mechanism isn't working when we expect it to be.

9.2.3 When to use browser.waitForAngularEnabled()

Knowing how and when to enable and disable waiting for Angular, even on pages that are part of an Angular app, is an important part of writing tests using Protractor. However, turning off waiting for Angular is a blunt instrument. It means your tests won't know when Angular is done updating the page, and you might have to resort to other synchronization methods. One such method is Expected Conditions.

9.3 Waiting with ExpectedConditions

When you tell Protractor not to wait for Angular, you might start seeing test failures if Angular updates the page while your test is running. It's tempting to just make the tests pass by sprinkling in `browser.sleep()` commands. This is a bad idea for a couple of reasons. First, the right amount of time to sleep is arbitrary and difficult to know ahead of time. It also slows down your tests, since you end up waiting a fixed amount of time, even if the condition you're waiting for has already occurred. Instead you can use `browser.wait()` and Expected Conditions to wait for specific conditions in your application to be true, like so

```
let EC = browser.ExpectedConditions;
browser.wait(EC.visibilityOf($('.popup-title'), 2000,
    'Wait for popup title to be visible.');
```

This will pause your test and repeatedly check whether the given condition is true, up to some specified timeout (2 seconds, in this case).

> **NOTE** You should always specify a timeout and an error message when using `browser.wait()`. If you don't specify a timeout, it will wait forever until your per-test timeout is hit, and having an error message makes timeouts much easier to debug.

Table 9.1 lists all the Expected Conditions built into Protractor. You can also combine any number of conditions with `and()`, `or()`, and `not()` like this

```
let EC = browser.ExpectedConditions;
let titleCondition =
  EC.and(EC.titleContains('foo'),
         EC.not(EC.titleContains('bar');
browser.wait(titleCondition, 5000,
  'Waiting for title to contain foo and not bar);
```

Table 9.1 Types of Expected Conditions

Name	When it's true
`alertIsPresent`	when an alert dialog is open.
`elementToBeClickable`	the given element is visible and enabled.
`textToBePresentInElement`	the element contains the specified text (case sensitive)
`textToBePresentInElementValue`	the element's 'value' attribute contains the specified text (case sensitive)
`titleContains`	document.title contains the given string (case sensitive)
`titleIs`	document.title exactly matches the given string
`urlContains`	the current URL contains the given string (case sensitive)
`urlIs`	the current URL exactly matches the given string
`presenceOf`	the element in present in the current page (but may or may not be visible)
`stalenessOf`	the element is no longer part of the page's DOM (the opposite of `presenceOf`)
`visibilityOf`	the element is present in the page, visible, and has a height and width greater than 0
`invisibilityOf`	the element is either not present in the DOM or is not visible (opposite of `visibilityOf`)
`elementToBeSelected`	the element is currently selected (if the element is an `<option>` or a checkbox or radio `<input>`)

Notice how you can combine expected conditions and assign them to variables so you can reuse them.

WARNING The `ExpectedConditions` object holds a reference to the `browser` object. Be careful using it in tests that restart or create multiple browsers, and use `browser.ExpectedConditions` instead of `protractor.ExpectedConditions`. Get the reference to ExpectedCondition after you restart or fork the browser.

9.3.1 Waiting for the contact list to load

Now that we know the basics of Expected Conditions, we have another way to make the test from listing 9.1 pass.

Listing 9.2 Using ExpectedConditions instead of waitForAngular

```
it('should be able to login', () => {
  let EC = browser.ExpectedConditions;
  browser.waitForAngularEnabled(false);
  browser.get('/assets/login.html');
  element(by.css('input.user')).sendKeys('username');
  element(by.css('input.password')).sendKeys('password');
  element(by.id('login')).click();

  const list = element(by.css('app-contact-list'));
  const listReady = EC.not(
    EC.textToBePresentInElement(list, 'Loading contacts'));  (1)
  browser.wait(listReady, 5000, 'Wait for list to load');    (2)
  expect(list.getText()).toContain('Jeff Pipe');
});
```

(1) Build the expected condition.
(2) Wait up to 5s for 'Loading contacts' to go away

Instead of turning on waitForAngular, we can wait for the 'Loading contacts' text to go away. There's no single right answer here – use whichever method is more readable and maintainable for your tests. Expected conditions are a helpful tool to have, especially when dealing with animations as we'll see in the next section.

9.3.2 Testing a dialog

Another common use of Expected Conditions is waiting for an animation to complete, perhaps when opening a dialog. Figure 9.2 shows a dialog from the contacts app. On the detail page for a contact, there's a button (circled in red) that opens a dialog that shows a feed of that contact's social media updates.

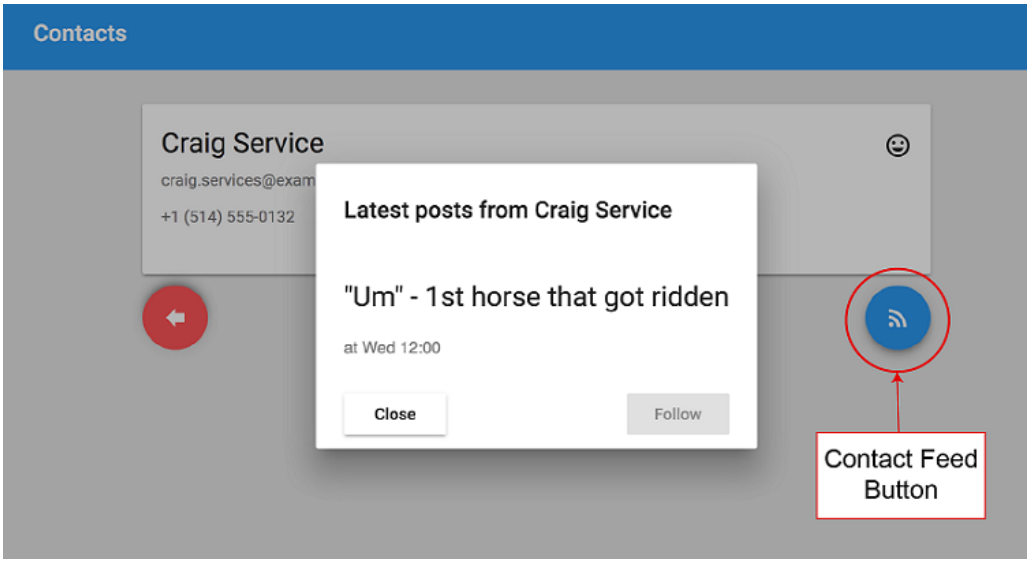

Figure 9.2 The social media feed dialog of a contact

When we click the feed button, the dialog animates opening. When we click "Close", it animates fading away briefly before closing. All these animations can be problematic when we try to test the dialog.

Listing 9.3 Testing the dialog with waitForAngular

```
describe('feed dialog', () => {
  beforeEach(() => {
    browser.get('/contact/4')
  });

  it('should open the dialog', () => {
    browser.waitForAngularEnabled(true);    ❶
    let feedButton = element(by.css('button.feed-button'));
    feedButton.click();                     ❷

    let dialogTitle = element(
      by.css('app-contact-feed h2.mat-dialog-title'));
    expect(dialogTitle.getText())
      .toContain('Latest posts from Craig Service');   ❸

    let closeButton = element(by.css('button[mat-dialog-close]'))
    closeButton.click();   ❹

    expect(dialogTitle.isDisplayed()).toBeFalsy();  ❺
  });
});
```

❶ Not needed, this is just to demonstrate that waitForAngular is turned on.

2. Click the feed button
3. We should see the title of the feed dialog
4. Close the dialog
5. The title should go away when the dialog closes.

This is a pretty simple test – it clicks the button to open the feed dialog, verifies that the expected title of the dialog is visible, then clicks the "Close" button. Unfortunately, when we run this test, this is the error we see:

```
1) contact detail feed dialog should open the dialog
  - Expected true to be falsy.
```

The last expectation fails because after we click the close button the dialog title is still visible while the closing animation runs. Even though Protractor is waiting for Angular to finish updating the page, it won't wait for the closing animation to end. This is the kind of situation where we might need to use Expected Conditions.

9.3.3 Waiting for elements to become stale

Let's try making that same test again, this time using Expected Conditions instead of relying on waitForAngular.

Listing 9.4 Testing the dialog with Expected Conditions

```
describe('feed dialog', () => {
  let EC;

  beforeEach(() => {
    browser.get('/contact/4');
    EC = browser.ExpectedConditions;
  });

  it('should open the dialog with expected conditions', () => {
    browser.waitForAngularEnabled(false);

    let feedButton = element(by.css('button.feed-button'));
    browser.wait(EC.elementToBeClickable(feedButton),
      3000, 'waiting for feed button to be clickable');  (1)
    feedButton.click();

    let dialogTitle = element(
      by.css('app-contact-feed h2.mat-dialog-title'))
    browser.wait(EC.visibilityOf(dialogTitle),
      1000, 'waiting for the dialog title to appear');   (2)
    expect(dialogTitle.getText())
      .toContain('Latest posts from Craig Service');

    let closeButton = element(by.css('button[mat-dialog-close]'))
    closeButton.click();
    browser.wait(EC.stalenessOf(dialogTitle),
      3000, 'wait for dialog to close');    (3)
    expect(dialogTitle.isPresent()).toBeFalsy();
  });
});
```

1. Wait for the feed button to be clickable.
2. Wait for the dialog title to be visible.
3. Wait for the dialog title to be removed from the page.

This test passes because we wait for the dialog title to become "stale" before the last expectation. In WebDriver tests, a "stale" element is one that we may have a reference to, but which was removed from the page. In this case, the title of the dialog box becomes stale when the closing animation finishes and the dialog is removed from the page. Elements that are removed from the page with `*ngFor` or `*ngIf` would also become stale.

This test disables waitForAngular and relies on Expected Conditions entirely, but you can also mix the two techniques. For example, we could have made the test from 9.3 pass simply by adding `browser.wait(EC.stalenessOf(dialogTitle))` before the expectation and left waitForAngular enabled. Either way is fine – the important thing is that your tests reliably do the same thing each time they run.

9.4 Creating custom conditions

Expected Conditions are powerful, but sometimes they might not be sufficient for our needs. Instead of just waiting for an element to be present or text to be visible, we might want to wait for a more complicated condition to be true. For example, we might need to wait until there's a certain number of elements that match a CSS selector. Or, perhaps the set of elements we're waiting for can't be described by single selector. In cases that are too difficult to express with Expected Conditions, we can use `browser.wait` with a custom condition.

9.4.1 Using browser.wait

The feed dialog from Figure 9.2 will automatically update with new posts from the contact. For testing purposes, it just shows a new update after a random delay, on average every 5 seconds. Let's say we add a feature that the "Follow" button will only be enabled when there are two or more posts from the contact. Here's a test that verifies that feature:

Listing 9.5 Using browser.wait with a custom condition

```
describe('feed dialog', () => {
  beforeEach(() => {
    browser.get('/contact/4');
  });

 it('should enable the follow button with more than two posts', () => {
    let feedButton = element(by.css('button.feed-button'));
    feedButton.click();

    let followButton = element(by.css('button.follow'))
    expect(followButton.isEnabled()).toBeFalsy();    (1)
    let moreThanOnePost = () => {
      return element.all(by.css('app-contact-feed mat-list-item')).count()
        .then((count) => {
          return count >= 2;      (2)
```

```
        })
    };
    browser.wait(moreThanOnePost, 20000, 'Waiting for two posts');    ❸

    expect(followButton.isEnabled()).toBeTruthy();    ❹
  });
});
```

❶ Verify the follow button is initially disabled.
❷ Count the number of md-list-items and return true if there's more than one.
❸ Wait until there to be two or more posts.
❹ Verify the follow button is enabled.

The first argument to `browser.wait` is a function that will be repeatedly run until either it returns true or the timeout is elapsed. This function can return a promise. In the example, we supply a function that looks for all elements that match the `'app-contact-feed mat-list-item'` selector, which will match each post in the feed. Because Protractor needs to send a request to the browser driver to inspect the page, the result of `element.all(…).count()` is a Promise with the number of elements that match the selector instead of just a number. We then chain this promise with a `.then()` block that returns a boolean which is true if the count is greater than or equal to two.

This is similar to how Expected Conditions work. The Expected Conditions that are built in to Protractor (from table 9.1) are just functions that inspect the page and return promises that are true when the condition is met. Listing 9.5 is an example of how you can create your own conditions in case you need them.

9.4.2 Getting elements from the browser

WebDriver helpfully converts DOM elements returned from the browser via a `browser.executeScript` call into instances of WebElement classes that you can use in your tests. So, instead of using element finders, you can write custom JavaScript that will run in the browser and return the elements you're looking for. Here's the test from listing 9.5, but using a custom element finder.

Listing 9.6 Retrieving elements with a custom finder

```
it('should enable the follow button (custom finder)', () => {
    let feedButton = element(by.css('button.feed-button'));
    feedButton.click();

    let followButton = element(by.css('button.follow'))
    expect(followButton.isEnabled()).toBeFalsy();

    function findAllPosts() {    ❶
      return document.querySelectorAll('app-contact-feed mat-list-item')
    }
    browser.wait(() => {
      return browser.driver.executeScript(findAllPosts)    ❷
        .then((posts: WebElement[]) => {
          return posts.length >= 2;
```

```
      })
  }, 20000, 'Waiting for two posts');

  expect(followButton.isEnabled()).toBeTruthy();
});
```

1. Function that runs in the browser and returns elements
2. Using the custom element finder in browser.wait

The test in Listing 9.6 is the same as the test in 9.5. However, instead of using an element finder, it uses a JavaScript function that runs in the browser and returns an array of WebElements. This example is somewhat trivial, but demonstrates how you can write custom JavaScript that extracts an arbitrary collection of DOM elements from the page. Remember that your tests are running in Node.js, but they can still execute JavaScript in the browser. The `findAllPosts()` function is run in the browser, but the result it returns can be used in your Node.js based Protractor tests.

9.5 Handling long-running tasks

The feed dialog in the contact detail page continuously updates with new posts. For demonstration purposes, this is done simply with an Observable that produces an infinite stream of random posts.

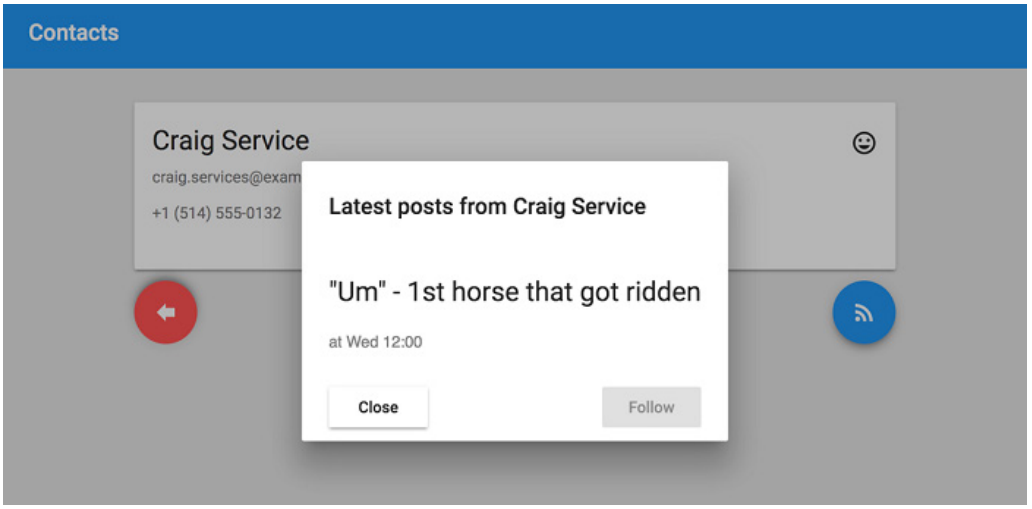

Figure 9.3 The contact feed dialog.

Listing 9.7 is the implementation of the feed dialog.

Listing 9.7 contact-feed.service.ts

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Rx';
import { FEED_UPDATES } from ‘./mock-updates’;

@Injectable()
export class ContactFeedService {
  constructor() { }

  public getFeed() {
    return Observable.interval(500)
      .map((x) => Math.random() * 2 + 2)
      .concatMap((x) => Observable.of(x).delay(x * 1000))  (1)
      .map((x) => FEED_UPDATES[Math.floor(Math.random() * FEED_UPDATES.length)]);     (2)
  }
}
```

(1) Transform the observable stream into a stream of randomly delayed events
(2) Randomly pick a string from FEED_UPDATES and put it in the stream

Where `FEED_UPDATES` is just an array of strings. The feed dialog component subscribes to this Observable, as seen in listing 9.8.

Listing 9.8 contact-feed.component.ts

```
import {Component, OnInit, OnDestroy, Optional, Inject} from '@angular/core';
import {MdDialogRef, MD_DIALOG_DATA} from '@angular/material';
import {ContactFeedService} from '../shared/services/contact-feed.service';
import {Subscription} from 'rxjs/Subscription';

@Component({
  selector: 'app-contact-feed',
  templateUrl: './contact-feed.component.html',
  styleUrls: ['./contact-feed.component.css']
})
export class ContactFeedDialogComponent implements OnInit, OnDestroy {
  sub: Subscription;
  updates: string[] = [];
  name: string;
  closeDisabled = true;

  constructor(public dialogRef: MdDialogRef<ContactFeedDialogComponent>,
              private feed: ContactFeedService,
              @Optional() @Inject(MD_DIALOG_DATA) data: any) {
    this.name = data.name;
  }

  ngOnInit() {
    this.closeDisabled = false;

    this.sub = this.feed.getFeed().subscribe((x) => {
      this.updates.push(x);
      if (this.updates.length >= 4) {
        this.updates.shift();
      }
    });   (1)
```

```
  }

  ngOnDestroy() {
    this.sub.unsubscribe();  ❷
  }
}
```

❶ Subscribe to feed updates and push them into the 'updates' property
❷ Clean up the subscription when the component is destroyed.

Unfortunately, our tests from listings 9.6 and 9.5 (which have `waitForAngular` enabled) will time out when trying to test this dialog

```
1) feed dialog should enable the follow button with more than two posts using executeScript
  - Error: Timeout - Async callback was not invoked within timeout specified by
      jasmine.DEFAULT_TIMEOUT_INTERVAL.
```

Protractor's waitForAngular hooks into the same method that Angular uses to run change detection and update template bindings. This is how Protractor knows that Angular is done updating the page, but it means that by default Protractor will wait until all asynchronous tasks that could update the page have finished. The contact feed dialog in our example polls forever, and thus Angular times out because there's always a pending task that can update the page.

9.5.1 Using Expected Conditions

We could just disable waiting for Angular. However, if we want to avoid flaky tests, this means we'll need to use Expected Conditions to wait after every action that can cause the page to update. That's what the test from Listing 9.4 did – here it is again as a reminder.

Listing 9.9 Testing the feed dialog with Expected Conditions

```
it('should open the dialog with expected conditions', () => {
  browser.waitForAngularEnabled(false); ❶
  let feedButton = element(by.css('button.feed-button'));
  browser.wait(EC.elementToBeClickable(feedButton),  ❷
    3000, 'waiting for feed button to be clickable');
  feedButton.click(); ❸

  let dialogTitle = element(by.css('app-contact-feed h2.mat-dialog-title'))
  browser.wait(EC.visibilityOf(dialogTitle),
    1000, 'waiting for close button to be clickable');
  expect(dialogTitle.getText()).toContain('Latest posts from Craig Service')

  let closeButton = element(by.css('button[mat-dialog-close]'))
  closeButton.click();  ❹
  browser.wait(EC.stalenessOf(dialogTitle), 3000, 'wait for dialog to close');
  expect(dialogTitle.isPresent()).toBeFalsy();
});
```

❶ Disabling waitForAngular.
❷ We need to wait for the initial page load.
❸ When we click the feed button, we'll need to wait for the dialog to show.

❹ Clicking the close button also requires a wait

Waiting after every action that could cause a page update can be tiresome, and it makes our test harder to read. It would be better if we could just write a test that used Protractor's automatic waitForAngular behavior, but doing so will require a bit of understanding about how browsers run asynchronous code, and how Angular knows when to update the page.

9.5.2 The browser event loop

To understand how Angular uses zones better, we first need to understand how JavaScript runs in the browser. JavaScript is single-threaded, meaning it's only ever doing one thing at a time. Somewhere inside your browser, there's an event loop that looks something like this:

```
while(true) {
  event = waitForNextEvent()
  doJavaScriptThings(event);
  doBrowserThings(event);
}
```

In this example `doBrowserThings()` refers to the work the browser does outside of your app's JavaScript – rendering the page, doing IO, etc. An event can be something like a timer firing, a mouse click event, or an XHR request changing in status. These events create "tasks" in the JavaScript VM, and there are three kinds of tasks: MicroTasks, MacroTasks, and EventTasks.

Table 9.2 Types of tasks

Task Type	When it runs
MicroTasks	Executed immediately, before the browser does any rendering or IO. `Promise.resolve()` will schedule a microtask.
MacroTasks	Guaranteed to run at least once and in the same order that they're scheduled. Macrotasks run interleaved with browser rendering and IO and are scheduled by `setTimeout` or `setInterval`. After a Macrotask finishes, all microtasks are executed before control passes back to the browser.
Events	Run in response to events (for example, `addEventListener('click', eventCallback)` or XHR state change). Unlike MacroTasks, Events might never be run.

All microtasks are run before control of the event loop goes back to the browser. Running a microtask might add more microtasks to the queue (for example, by making a `Promise.resolve()` call). Once the microtask queue is empty, control passes back to the browser so it can render the page, perform IO, and wait for the next Event or Macrotask to occur. So really, the situation is a little more complicated than the simple `while(true)` loop from earlier. Now that we know a bit more about how browsers work, let's consider how this relates to Angular.

9.5.3 What happened to $timeout?

If you've used AngularJS, you might remember the `$timeout` service. When doing asynchronous work in AngularJS, instead of using `window.setTimeout()` or `XMLHttpRequest()` directly, you needed to use the special AngularJS service `$timeout` and `$http`. These services were wrappers around the native browser calls that would make sure change detection ran after the asynchronous task was done, so that the content of your page would update if your model changed.

In Angular, these special services are no longer needed. Instead, Angular uses a library called Zone.js to run your application's asynchronous tasks in a context called the "Angular Zone". A "Zone" is an execution context that persists across async tasks – sort of like thread-local storage in Java, but for async tasks. Zone.js provides this functionality by patching all the browser APIs that create async calls with hooks that track which zone that task is running in. This is how Angular knows when an async callback started by your app occurs and is able to run change detection after it, thus removing the need for `$timeout`.

9.5.4 Highway to the Angular Zone

Protractor knows about the Angular Zone: it knows when there are tasks pending that might cause a change detection. When you enable waitForAngular, Protractor will cause all of your WebDriver commands to wait until there are no more tasks pending in the Angular Zone. Let's look at a simple example of running asynchronous tasks in a browser.

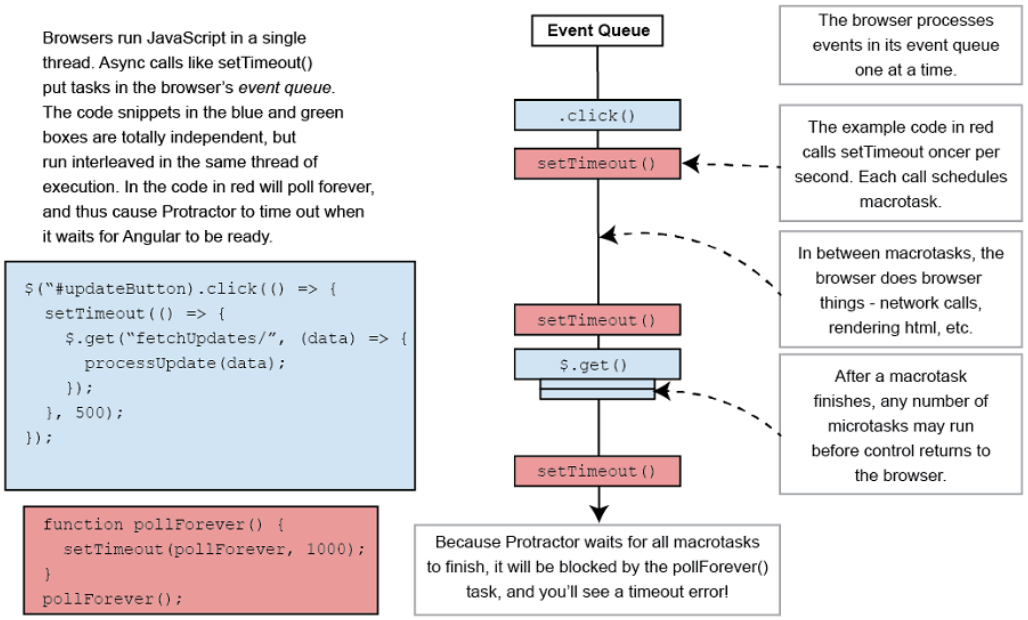

Figure 9.4 Async tasks running in a browser

Assume that the code in Figure 9.4 is running in the Angular Zone. In this example, Protractor would wait forever, because the `pollForever()` function (from the red box) is constantly creating a task in the Angular Zone. If we want to avoid this fate, we can move that code inside a call to `NgZone.runOutsideAngular()`. Doing so will mean that `pollForever()` will no longer trigger a change detection when it runs, and also that Protractor will no longer wait for it to finish.

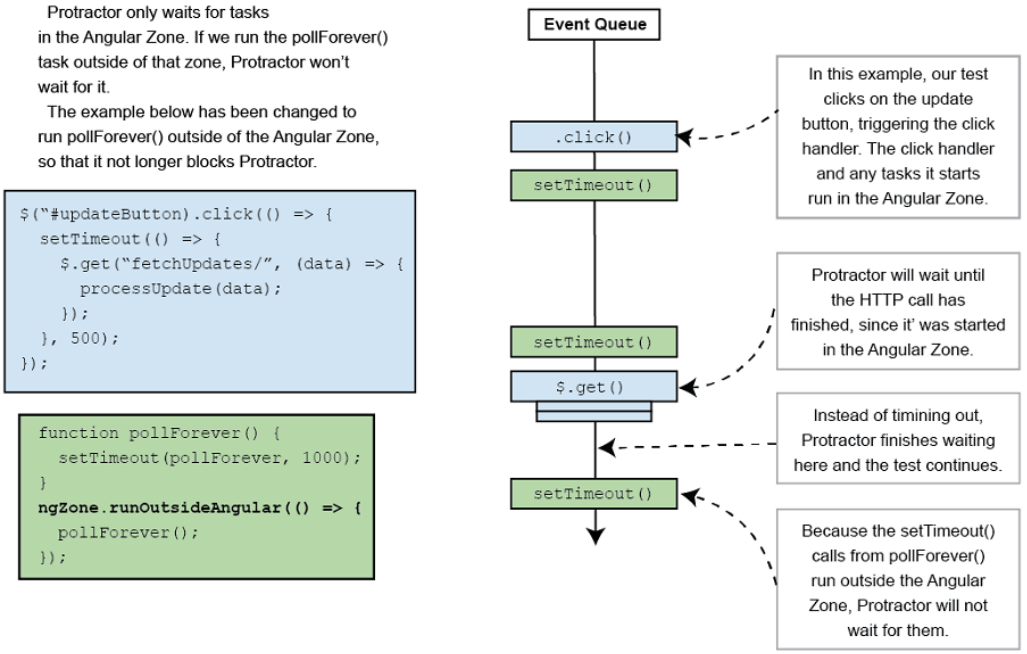

Figure 9.5 Running a polling task outside the Angular Zone.

Now we have our long running task running in a way that no longer causes Protractor to wait forever. Let's apply this same technique to the Contacts app example and use it to fix the test.

9.5.5 Fixing the test

Now we know why our test was timing out earlier. The updates from the `ContactFeedService` were scheduling async tasks in the Angular zone, and because it's a continuous stream of tasks Protractor will wait indefinitely until the test times out. We could disable waitForAngular and use Expected Conditions, but we can also fix the test by changing the `ContactFeedComponent`.

Listing 9.9 Using runOutsideAngular in ContactFeedDialogComponent

```
import {Component, OnInit, OnDestroy, NgZone, Optional, Inject} from '@angular/core';
import {MdDialogRef, MD_DIALOG_DATA} from '@angular/material';
import {ContactFeedService} from '../shared/services/contact-feed.service';
import {Subscription} from 'rxjs/Subscription';

@Component({
  selector: 'app-contact-feed',
  templateUrl: './contact-feed.component.html',
  styleUrls: ['./contact-feed.component.css']
})
```

```
export class ContactFeedDialogComponent implements OnInit, OnDestroy {
  sub: Subscription;
  updates: string[] = [];
  name: string;
  closeDisabled = true;

  constructor(public dialogRef: MdDialogRef<ContactFeedDialogComponent>,
              private feed: ContactFeedService,
              private zone: NgZone,     (1)
              @Optional() @Inject(MD_DIALOG_DATA) data: any) {
    this.name = data.name;
  }

  ngOnInit() {
    this.closeDisabled = false;

    this.zone.runOutsideAngular(() => {     (2)
      this.sub = this.feed.getFeed().subscribe((x) => {
        this.zone.run(() => {               (3)
          this.updates.push(x);
          if (this.updates.length > 4) {
            this.updates.shift();
          }
        });
      });
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

(1) Inject NgZone into the component.
(2) Run the subscription outside the Angular zone, so it won't block Protractor.
(3) Add the update in the Angular zone, so the change propagates to the page.

Now that the subscription is created outside of the Angular zone, it will no longer block Protractor. However, we need to apply the update within the Angular zone so that Angular will know about the change to our model and will update the page. With only a small change to the component, our test passes! Protractor will still wait while each update is rendered in the dialog, but will no longer time out waiting for the stream of updates to finish.

9.6 Summary

We've covered a wide range of subjects, from how Protractor controls a browser during a test to how Angular detects changes in your model and updates the page. In this chapter, you've learned the following:

- Browser tests are essentially multi-threaded programs. Protractor synchronizes with your application by waiting for Angular to be done updating the page.
- Sometimes you need to disable waiting for Angular in your tests.

- Use Expected Conditions instead of `browser.sleep`. Tests are more reliable when they wait for a specific condition to be true, rather than just pausing for an arbitrary amount of time. Protractor has many Expected Conditions you can use out of the box.
- If there isn't an Expected Condition that meets your needs, you can make your own using `browser.wait`.
- Angular uses Zone.js to watch for async tasks that might cause the page to change. Protractor also uses Zone.js to wait until every async task that might modify the page has finished.
- Sometimes you'll need to change your application to run certain async tasks outside the Angular zone, otherwise you might end up unintentionally blocking Protractor from testing your page.

10

Advanced Protractor topics

This chapter covers:

- Protractor configuration files
- Screenshot testing
- Debugging tests

Protractor is a powerful tool, and like any powerful tool there are always more creative ways to use it than are written in the manual. As developers working on Protractor, we've found there are common scenarios that people tend to ask about. Some common questions are "How do I extend Protractor's behavior with plugins?" or "How do I create screenshots when my tests fail?" We've collected our best tips for working with Protractor in this chapter to help you get the most out of your end-to-end tests! The examples in this chapter can be found at

https://github.com/testing-angular-applications/testing-angular-applications/tree/master/chapter10

10.1 Configuration file in depth

Protractor provides many knobs and levers that change how it launches based on configuration options. In chapter 8, we used a single way to run Protractor tests. We launched the Chrome browser directly using the Jasmine test framework against a list of TypeScript spec files. This configuration setup is reasonable for testing locally, but what if you need to test on a different browser or a set of browsers? How can we set up tests to run in a continuous integration environment without a GUI?

10.1.1 Driver Provider Options

In the Protractor examples in chapter 8, we used the `directConnect` flag to launch Chrome with the ChromeDriver binary. There are other ways that Protractor can launch a new browser instance to run tests. In the Protractor repository, these WebDriver options are referred to as *driver providers*. Different driver providers can be set with the configuration settings listed in table 10.1.

Table 10.1: Driver Provider Options

Setting	Comments
`direct connect`	Launching a browser with `directConnect` is great while developing your Protractor test suite, since it allows you to work without needing to start up a selenium standalone server. Using `directConnect` is limited to launching tests for the Chrome with the ChromeDriver binary and Firefox with the GeckoDriver binary. The typical way to download just these binaries is to use the webdriver-manager npm package. To get these binaries, run: `webdriver-manager update --standalone false`. Specifying "`—standalone false`" prevents webdriver-manager from downloading the selenium standalone server jar file. By default, Protractor uses the binaries found in the webdriver-manager npm's folder. However, if you need to bypass the webdriver-manager's downloaded binaries, you can specify the path to your driver in the configuration file with the `chromeDriver` and `geckoDriver` options.

`selenium standalone server`	Setting `seleniumAddress` will tell Protractor to start browsers using the Selenium standalone server. Using the Selenium standalone server allows you to run tests that control browsers running on a different machine. This is helpful if the machine you are currently using is headless or without a desktop environment.
	The typical value for the `seleniumAddress` is `'http://127.0.0.1:4444/wd/hub'` when launching the standalone server locally using the default port.
	Typically, you just run `webdriver-manager update` and then launch the selenium standalone server with `webdriver-manager start`. If you need more control over Selenium you can launch it manually like so:
	`java -Dwebdriver.gecko.driver=/tmp/geckodriver`
	`-Dwebdriver.chrome.driver=/tmp/chromedriver`
	`-jar /path/to/selenium-server-standalone.jar`
	`-port 4444`

`Browser Stack`	BrowserStack is a cloud service for selenium testing on desktop and mobile browsers. To use BrowserStack, set the browserstackUser and browserstackKey options in the configuration file. BrowserStack is a paid service and is out of scope of this book; however, more information can be found on their website: https://www.browserstack.com/.
`Sauce Labs`	Sauce Labs is another cloud service for selenium testing on desktop and mobile browsers. Configure Sauce Labs by setting the sauceUser and sauceKey in the configuration file. Sauce Labs is another paid service and is also out of scope of this book; however, more information can be found at: https://saucelabs.com/.
`Specifying the selenium server jar file` **OR** **Local driver option: no driver provider options passed in configuration file**	If you set seleniumServerJar or don't set any driver providers in the config file, Protractor will start and shut down the selenium server for you. If the selenium server jar is not defined in the configuration file, Protractor will uses binaries downloaded to your project's node modules webdriver-manager/selenium directory. To download the binaries, you need to run `webdriver-manager update`. After you have these binaries downloaded, running Protractor will also launch the selenium server locally. When the path to the selenium server jar is defined in the configuration file, it will use the absolute path to launch the selenium server. In addition to the specifying the selenium jar, you can also specify the `chromeDriver` path. Similarly, for tests against the Firefox browser, you could specify the `geckoDriver` path. If the paths are not set for either `chromeDriver` or `geckoDriver`, it will use the default locations specified by the locally installed webdriver-manager module.

We should note that there's an order of precedence for multiple driver providers set in the same configuration file. The order of precedence is the same as the order shown in Table 10.1: direct connect, selenium standalone server, Browser Stack, Sauce Labs, a specified selenium server jar file, and then finally, local launch. In this section we've hinted that there are ways to use browsers other than Chrome. In the next section on capabilities, we show how to run tests against other browsers.

10.1.2 Desired capabilities

Let's say we want to test against Firefox instead of Chrome. All we need to do is change the `browserName` in our Protractor config, as shown in Listing 10.1. We could also launch with driver providers: seleniumAddress, seleniumServerJar or local driver option. In addition to Firefox, we can also use other browsers like: Safari, Microsoft Edge, or Internet Explorer. Only Chrome and Firefox support directConnect, so for other browsers you should set `seleniumAddress`.

Listing 10.1 Specifying capabilities for Firefox

```
exports.config = {
  directConnect: true,
  capabilities: {
    browserName: 'firefox'
  }
};
```

> **TIP** The only caveat when using Safari is that we need to run it on an OSX device, which comes bundled with the Safari driver. When running Safari, you will also need to turn on "Allow Remote Automation" under the "Develop" menu. If we want to run on Microsoft's Edge and Internet Explorer browsers, we'll have to use a Windows machine. Both browsers require their own matching browser drivers. When using Internet Explorer, we can download the driver using the command `webdriver-manager update --ie` and start the selenium standalone server with command `webdriver-manager start --ie`.

Beyond simply changing the browser name, we can also change other browser settings. However, if we set a desired capability that isn't implemented by the browser driver, we won't see an error during the test. Thus, when using a browser driver, it's important to check that we're using supported capabilities.

One good use case for desired capabilities is running Chrome in *headless mode*, which does not require a desktop UI. Headless Chrome is available in versions 59+ (60+ for Windows machines). Because we're running Chrome without a user interface, being able to log network traffic also helps validate that the test is working. With network traffic enabled, we might want to extend our test to track how long each JavaScript dependency takes to load. In listing 10.2, we set the headless `chromeOption` as well as `loggingPrefs`. When we enable performance logging in `loggingPrefs` to true, we'll be able to access each browser session's log in an `afterEach` method.

Listing 10.2 Configuration using headless Chrome – test_capabilities/protractor.conf.js

```
exports.config = {
  directConnect: true,
  capabilities: {
    browserName: 'chrome',
    chromeOptions: {                                    (1)
      args: ['--headless', '–disable-gpu']              (2)
    },
    loggingPrefs: {                                     (3)
      performance: 'ALL',
      browser: 'ALL'
    }
  },
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  specs: ['e2e/**/*.e2e-spec.ts'],
  onPrepare: () => {
    require('ts-node').register({
      project: 'e2e'
    });
  },
```

```
  useAllAngular2AppRoots: true
};
```

1. Set headless option in chromeOptions
2. Disable GPU arg is temporarily required. See https://developers.google.com/web/updates/2017/04/headless-chrome
3. Set performance logging for each browser session.

In listing 10.3, after every test, we check that browser is getting traffic and log that information to the console.

Listing 10.3 Test using headless Chrome - test_capabilities/e2e/test.e2e-spec.ts

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

describe('listing example', () => {
  it('load /', () => {
    console.log('get /')
    browser.get('/');
    expect(browser.getCurrentUrl()).toEqual(browser.baseUrl + '/');
  });

  it('click "+" button -> /add', () => {
    console.log('click "+" button -> /add')
    element(by.id('add-contact')).click();
    expect(browser.getCurrentUrl()).toEqual
    (browser.baseUrl + 'add');
  });

  afterEach(() => {                                                    (1)
    browser.manage().logs().get('performance').then((browserLogs) => {
      expect(browserLogs).not.toBeNull();                              (2)
      browserLogs.forEach((browserLog) => {
        let message = JSON.parse(browserLog.message).message;
        if (message.method == 'Network.responseReceived') {
          if (message.params.response.timing) {                        (3)
            let status = message.params.response.status;
            let url = message.params.response.url;
            console.log('status=' + status + ' ' + url);
          }
        }
      });
    });
  });
});
```

1. Log the performance values from the browser logs after each test
2. Check to see if traffic is going to the browser
3. If the response is valid, log the response code and URL to console

Instead of using a single set of capabilities to launch a browser, what if you want to run the exact same test against other browsers? To launch against multiple browsers, you will need to use `multiCapabilities`. You can specify `multiCapabilities` as an array of desired

capabilities as shown in listing 10.4. Using multiple capabilities requires running against a Selenium server, so we need to set the `seleniumAddress` in the configuration file.

Listing 10.4 Multicapabilities – test_multicapabilities/protractor-chrome.conf.js

```
exports.config = {
  multiCapabilities: [ {
    browserName: 'chrome'                                    (1)
  }, {
    browserName: 'chrome'                                    (1)
  } ],
  seleniumAddress: 'http://127.0.0.1:4444/wd/hub',
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  specs: ['e2e/**/*.e2e-spec.ts'],
  useAllAngular2AppRoots: true,
  onPrepare: () => {
    require('ts-node').register({
      project: 'e2e'
    });
  }
};
```

(1) Running the specs in Chrome twice

It's important to consider how many tests should run in parallel browsers. If we run our tests with multiple browsers, we could run into CPU- or RAM-resource limitations. Also, the main reason to run tests over a set of browsers is to validate that our app is compatible with those browsers.

For example, imagine some of our users have noticed navigation issues a particular feature with Microsoft Edge. We might want to use a subset of tests to validate this feature with both Chrome and Microsoft Edge browsers.

10.1.3 Plugins

Protractor has a concept of *lifecycle hooks* during the test execution. Lifecycle hooks allow us insert custom functionality at different points during test execution. Some lifecycle hooks can gather test results or modify the test output. Protractor lets you use these lifecycle hooks by adding plugins to the configuration file. We used a plugin in chapter 8, when we specified an `onPrepare` so we could load TypeScript spec files using the ts-node npm package.

Another good use of plugins is to create custom report artifacts. A typical report artifact is an xUnit report, which is a test report format defined in the Junit test framework. In order to create JUnit-style test reports, you'll need to override Jasmine's reporter with the jasmine-reporters node module shown in listing 10.5.

Listing 10.5 JUnit-style reports using onPrepare method – test_plugins/protractor-onprepare-method.conf.js

```
exports.config = {
  directConnect: true,
  capabilities: {
    browserName: 'chrome'
  },
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  specs: ['e2e/**/*.e2e-spec.ts'],
  onPrepare: () => {
    let jasmineReporters = require('jasmine-reporters');
    let junitReporter = new jasmineReporters.JUnitXmlReporter({

      savePath: 'output/',                                          (1)
          consolidateAll: false                                     (2)

    });
    jasmine.getEnv().addReporter(junitReporter);                    (3)
    require('ts-node').register({
      project: 'e2e'
    });
  },
  useAllAngular2AppRoots: true
};
```

(1) The relative path to save the JUnit style reports
(2) If true, aggregate test results; if false, create files
(3) Override the Jasmine default reporter with the new junitReporter

Another way to use plugins is by setting the plugins configuration setting. The plugins configuration setting takes an array of objects in which each object defines a plugin. We can rewrite our onPrepare method as a plugin. In listing 10.6, we show the same example as in 10.5, but this time using a plugin configuration.

Listing 10.6 JUnit-style reports using plugins configuration setting – test_plugins/protractor-onprepare-plugin.conf.js

```
exports.config = {
  directConnect: true,
  capabilities: {
    browserName: 'chrome'
  },
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  specs: ['e2e/**/*.e2e-spec.ts'],
  plugins: [{
    inline: {
      onPrepare: () => {
        let jasmineReporters = require('jasmine-reporters');
        let junitReporter = new jasmineReporters.JUnitXmlReporter({
          savePath: 'output/',
          consolidateAll: false
        });
        jasmine.getEnv().addReporter(junitReporter);
        require('ts-node').register({
```

```
          project: 'e2e'
        });
      }
    }
  }],
  useAllAngular2AppRoots: true
};
```

Until now, all our tests have run on a single machine; however, we might need to change the behavior depending on our testing environment. In some situations, for example, we might not want to use create JUnit style reports using plugins. In the next section we'll look in depth at how to set environment-specific configuations.

10.1.4 Environment variables

In previous sections, we created configuration files with different options. For example, you might have Chrome installed on one machine and Firefox on another machine. Or on some environments, you might need to produce JUnit style reports, while in others you might not want to have reports generated.

How can you change Protractor's behavior based on the environment in these use cases? One way is by setting environment variables. In listing 10.6, we set environment variables to determine which browser the test will use, and whether to use direct connect or selenium standalone server. Remember that if you set the DIRECT_CONNECT to true, and the SELENIUM_ADDRESS to http://127.0.0.1:4444/wd/hub", the test will launch with direct connect based on how Protractor handles these driver providers.

> **INFO** If you export the variables in the bash terminal session, the variables will exist only for that terminal session. If you need these variables to persist beyond the terminal session, you can set them in your ~/.bash_profile on OSX or Linux.

Listing 10.7 Setting environment variables

```
export BROWSER_NAME='chrome'
export DIRECT_CONNECT=true                    (1)
export SELENIUM_ADDRESS=''                    (2)
```

(1) Use direct connect when set to true
(2) Use selenium standalone server if defined and if direct connect is false

Now that the environment variables are exported, you can modify the Protractor configuration file to change behavior based on them. In listing 10.7, we set `directConnect` and `seleniumAddress` based on environment variables, which we can read from node using `process.env`. If `process.env.DIRECT_CONNECT` and `process.env.SELENIUM_ADDRESS` are not defined, Protractor will launch the Selenium standalone server using a local driver. When you set the `browserName`, if you don't set `process.env.BROWSER_NAME`, Protractor will default to using Chrome.

Listing 10.8 Using environment variables -- test_environemnt/protractor.conf.js

```
exports.config = {
  directConnect: process.env.DIRECT_CONNECT,                          (1)
  seleniumAddress: process.env.SELENIUM_ADDRESS,                      (2)
  capabilities: {
    browserName: (process.env.BROWSER_NAME || 'chrome')               (3)
  },
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  specs: ['e2e/**/*.e2e-spec.ts'],
  onPrepare: () => {
    if (process.env.BROWSER_NAME == 'chrome') {                       (4)
      let jasmineReporters = require('jasmine-reporters');
      let junitReporter = new jasmineReporters.JUnitXmlReporter({
        savePath: 'output/',
        consolidateAll: false
      });
      jasmine.getEnv().addReporter(junitReporter);
    }
    require('ts-node').register({
      project: 'e2e'
    });
  },
  useAllAngular2AppRoots: true
};
```

(1) Use direct connect if the DIRECT_CONNECT environment variable is true
(2) Use a selenium address if the SELENIUM_ADDRESS is not equal to an empty string
(3) Use the browser environment variable set in BROWSER_NAME
(4) If the browser is chrome, create JUnit style reports

Instead of setting environment variables, you could also create separate Protractor configuration files. Although having multiple Protractor configuration files for each environment might be an easy solution, it requires maintenance if you need to make a change that affects all the configuration files.

Now that we know how to configure Protractor in depth, let's use this knowledge to create a new kind of test. In the next section, we create a custom plugin and config that will let us compare browser screenshots in our tests.

10.2 Screenshot testing

We write tests to prevent our mistakes from becoming user-facing issues. Up until now, we've been testing only the logic of our application. However, the appearance of our web app is just as important. One way to verify that the appearance of our app is correct is to have a test that fails when a screenshot changes.

You might think that such a test would be fragile, and you'd be right. This test will fail whenever the look of our app changes, intentionally or not. However, it's a useful safeguard to catch unintentional CSS regressions, which can be easy to introduce and hard to check. After all, who really wants to spend all day before a release clicking through each page in our app, verifying that no awkward CSS errors have slipped through? The test will guard against

unintentional style errors, but the price is that we need to update the test whenever we intentionally change our app's CSS.

10.2.1 Taking screenshots

Taking screenshots in Protractor is easy; we can call `browser.takeScreenshot()` to take a screenshot. Unlike the other WebDriver commands we've seen so far, `takeScreenshot()` returns a promise. As mentioned in earlier chapters, a promise represents the future value of an asynchronous operation. Don't worry if you're not familiar with promises; we'll cover them in the next section. The important thing to know is that to get the screenshot image, we need to call `.then()` on the result of `takeScreenshot()` and pass it a function that does something with the data. The following listing is an example of taking a screenshot in a test.

Also unlike other WebDriver commands, Protractor will not automatically wait for Angular before executing `takeScreenshot()` – it will take a screenshot immediately, which is useful when using screenshots for debugging. Thus, if we want to take the screenshot after Angular is done updating the page, we'll have to manually call `browser.waitForAngular()`.

Listing 10.9 test_screenshot/e2e/screenshot.e2e-spec.ts

```
describe('the contact list', () => {
 beforeAll(() => {
   browser.get('/');
   browser.driver.manage().window().setSize(1024,900); (1)
 });

 it('should be able to login', (done) => { (2)
   const list = element(by.css('app-contact-list'));
   browser.waitForAngular();    (3)

   browser.takeScreenshot().then((data) => {
     fs.writeFileSync('screenshot.png', data, 'base64');
     done();  (4)
   })
 });
});
```

(1) Set the window size before taking a screenshot
(2) Add the 'done' parameter to our function definition.
(3) Wait for the page to load before taking a screenshot.
(4) Let Jasmine know the test is done.

This test is different from the previous Protractor tests in a couple of ways. The takeScreenshot() command returns a promise. In fact, all WebDriver commands return promises, but Protractor has some hidden magic that lets you ignore that and write your tests as if they were synchronous (we'll cover that in more detail in the next section).

The other important difference is that the test is now asynchronous; it needs to wait until the screenshot is written to disk before finishing. We can make any Jasmine test asynchronous by accepting a 'done' callback in the function we define for our `it()` block. Jasmine will wait

until we execute that callback before finishing the test. This allows us to write tests with asynchronous behavior, like calling `setTimeout()` or making network calls – or in this case, waiting for the screenshot data.

10.2.2 Taking screenshots on test failure

Previously, we saw how we can use plugins to extend Protractor's behavior. The plugin API provides hooks that you can use to add custom logic to your Protractor tests. Let's use the plugin API to take a screenshot of the browser when a test fails. First, we can add a plugin to our test's config by adding a plugin section, like this:

```
plugins: [{
  path: './screenshot_on_failure_plugin.ts'
}],
```

This loads the plugin, which we define in a separate TypeScript file. Protractor plugins define functions that are called at different points in the test process. In our case, we simply need to define a `postTest()` function. Protractor calls the `postTest`() function after each it() block finishes. The function receives two arguments – whether the test passed, and an object containing a description of the test. If the test fails, we take a screenshot and save it to a file based on the name of the test that failed. This produces a screenshot at the moment of failure for each failing test in our test suite.

Listing 10.10 test_screenshot/screenshot_on_failure_plugin.ts

```
import {browser} from 'protractor';
import * as fs from 'fs';

export function postTest(passed: boolean, testInfo: any) {      (1)
 if(!passed) {
   const fileName = `${testInfo.name.replace(/ /g, '_')}_failure.png` (2)
   return browser.takeScreenshot().then((data) => {
     fs.writeFileSync(fileName, data, 'base64')  (3)
   });
 }
}
```

(1) Call this function after each it() block.
(2) Test name as the filename for the screenshot
(3) Write new screenshot synchronously to disk

You can find this plugin and config file in the Chapter 10 code repo on GitHub (https://github.com/testing-angular-applications/testing-angular-applications/tree/master/chapter10/test_plugins). Try it for yourself - make a failing test, and verify that you get a screenshot of the browser at the time of failure. If you want to learn more about writing plugins, check out the official docs in the Protractor repo (github.com/angular/protractor/blob/master/docs/plugins.md). This simple plugin will help

debug why a test failed, but we can also use screenshots as part of our tests, as you'll see in the next section.

10.2.3 Comparing screenshots

It can be hard to make an automated test that verifies the appearance of an application. However, we can make a simple screenshot test that will fail when there are any major, unintended changes in our application's appearance. We can use the 'looks-same' npm package to compare a screenshot of the browser against a reference image.

Listing 10.15 shows a couple of helper functions that use the 'looks-same' library to compare a screenshot to a reference image. The `writeScreenshot()` function simply encapsulates some of the boilerplate around writing a screenshot to disk. The `compareScreenshot()` function takes the *callback-oriented* API of looks-same and wraps it in a promise that's resolved with the value of the screenshot comparison.

The `compareScreenshot()` helper has a couple of useful features. First, we can easily update the golden image if we set the environment variable `UPDATE_SCREENSHOT`. For example, if we run our tests with `UPDATE_SCREENSHOT=1` protractor screenshot_test.conf.js, the test will run, but instead of comparing the screenshot to the reference, it overwrites the reference image with the new screenshot. We can then commit this new image to git and use it as the reference in future tests.

If the images are different, `compareScreenshot()` automatically calls `looksSame.createDiff()`. This creates an image showing the difference between the current screenshot and the reference image, so we can easily see what went wrong.

Listing 10.11 test_screenshot/e2e/screenshot_helper.ts

```
function writeScreenshot(data) { (1)
  return new Promise<string>(function (resolve, reject) {
    const folder = fs.mkdtempSync(`${os.tmpdir()}${path.sep}`);
    let screenshotFile = path.join(folder, 'new.png');
    fs.writeFile(screenshotFile, data, 'base64', function (err) {
      if (err) {
        reject(err);
      }
      resolve(screenshotFile);
    });
  });
}

export function compareScreenshot(data, golden) {
 return new Promise((resolve, reject) => {
   return writeScreenshot(data).then((screenshotPath) => {
     if (process.env['UPDATE_SCREENSHOTS']) {  (2)
       fs.writeFileSync(golden, fs.readFileSync(screenshotPath));
       resolve(true);
     } else {
       looksSame(screenshotPath, golden, {}, (error, equal) => {
         if (!equal) {
           looksSame.createDiff({  (3)
```

```
            reference: golden,
            current: screenshotPath,
            diff: 'diff.png',      ❹
            highlightColor: '#ff00ff'
          }, function (error) {
            resolve(equal);
          });
        } else {
          resolve(equal);
        }
      })
    }
  });
});
}
```

❶ helper function that writes the screenshot to disk.
❷ We can use an environment variable to control the helper
❸ If the screenshot is different, create an image highlighting those differences.
❹ The difference image is written to 'diff.png'

Listing 10.16 shows a test that uses these helpers. Note that in the test we need to explicitly call `browser.waitForAngular()`. Usually, when we turn on for waiting for Angular with `browser.waitForAngularEnabled(true)`, Protractor waits for Angular to be ready before executing each WebDriver command. However, screenshots are taken immediately, so we'll need to manually wait for the contact list to load.

We call `then()` on a promise to handle the result – the callback we pass to `then()` will be invoked when the asynchronous operations the promise represents are finished. If, when handling the result of a promise, we need to make a new asynchronous call, we can simply return a promise and add another `then()` block. This is called *promise chaining*, because we chain our then() blocks together, one per asynchronous operation. In listing 10.16, we chain together two asynchronous operations – taking the screenshot and comparing the screenshot to the reference – so we have two `then()` blocks.

Listing 10.12 An example screenshot test

```
it('should be able to login', (done) => {
  const GOLDEN_IMG = path.join(__dirname,'contact_list_golden.png'); ❶
  const list = element(by.css('app-contact-list'));
     browser.waitForAngular();

  browser.takeScreenshot()
    .then((data) => {
      return compareScreenshot(data, GOLDEN_IMG);  ❸
    })
    .then((result) => {
      expect(result).toBeTruthy();  ❹
      done();
    })
});
```

1. We save the golden image in the same directory as the test, so we can check it in to git.
2. Return the promise from compareScreenshot to chain it into the next then() block.
3. Result will be true if the screenshots matched.

For example, our contacts list might look like figure 10.6.

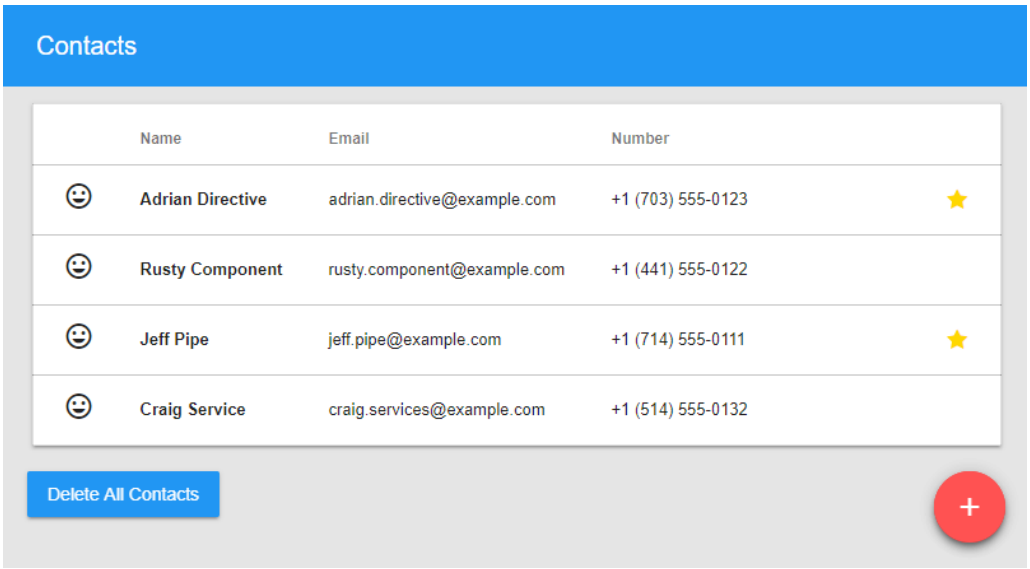

Figure 10.1 The reference screenshot image

Suppose we accidentally broke our application's CSS by adding the following to contact-list.component.css:

```
.add-fab {
  float: right;
  cursor: pointer;
  position: absolute;
}
```

Now, the list looks like figure 10.7.

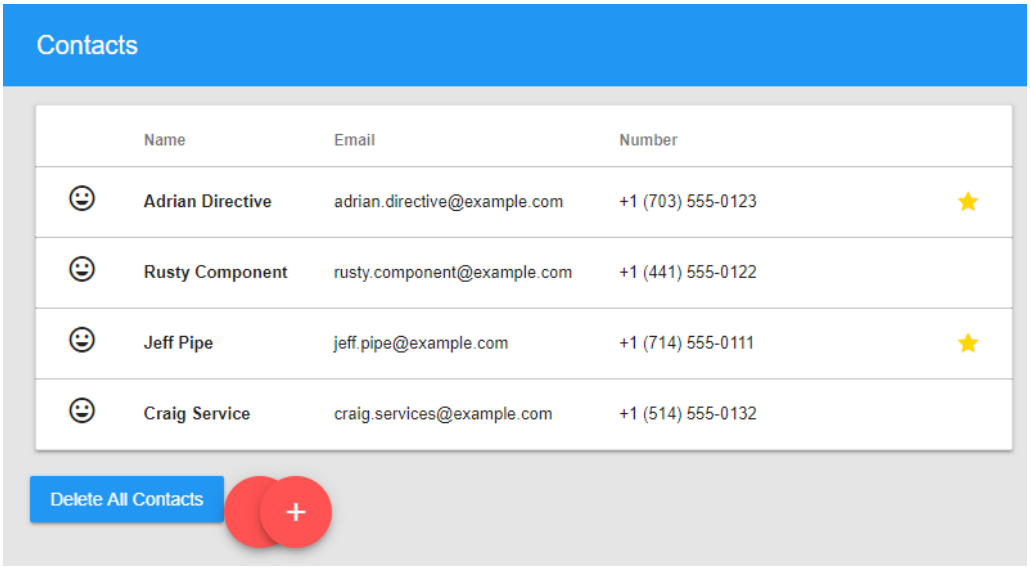

Figure 10.2 The contact list with broken CSS

Our tests might still pass, despite the page being obviously broken. However, our screenshot test will fail and produce this diff image.

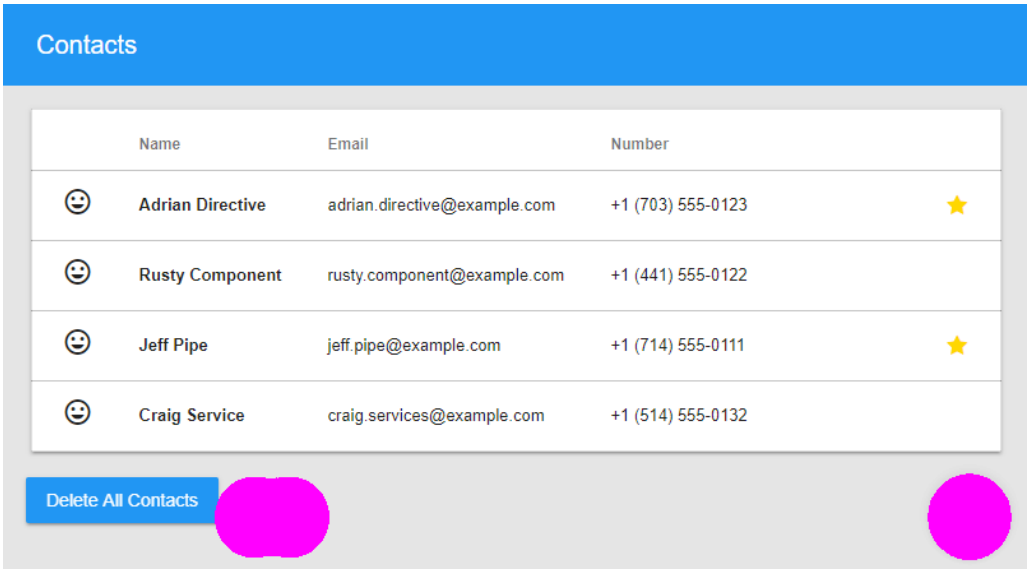

Figure 10.3 The difference image highlights where the current screenshot differs from the reference.

Notice how the diff image highlights the part of the screenshot that changed. You can change the highlight color if pink isn't your thing. The important part is that when our screenshot test fails, we can simply check the diff image to see what went wrong. If the change is expected, we can rerun the test with `UPDATE_SCREENSHOT=1` to update the reference image.

10.3 Experimental debugging features

It can be hard to know what might have caused a Protractor test to fail. Fortunately, there are some experimental features recently added to Protractor that can make it easier to debug failing tests. In this section, we'll use these new features to debug a test in the `chapter10/test_experimental` directory of the book's repository (which, as mentioned, can be found at https://github.com/testing-angular-applications/testing-angular-applications).

Here's the test we'll be working with in this section.

Listing 10.13 test_experimental/e2e/add-contact.e2e-spec.ts

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

describe('contact list', () => {
  beforeAll(() => {
    browser.get('/');
  });

  it('should be able to add a contact', () => {
    element(by.id('add-contact')).click();   (1)

    element(by.id('contact-name')).sendKeys('Ada Contact');     (2)
    element(by.css('.create-button')).click();                  (2)

    expect(element(by.css('app-contact-list')).getText())       (3)
          .toContain('Ada Contact');                            (3)
  });
});
```

(1) Click the add contact button.
(2) Type in a name for the contact and click the create button.
(3) Verify the new contact shows in the contact list.

This is a simple test, but because it moves between two different pages (the contact list and the add contact view), it can be a little difficult to debug when things go wrong. Let's look at some tools to help with that.

10.3.1 WebDriver Log

When a Protractor test runs, it sends commands to the browser telling it what to do. These are commands like "find an element", "click on an element", or "get the text of an element. Even if you watch the browser as your test runs, it can be hard to know exactly what's going on.

Instead, you can have Protractor create a log of the WebDriver commands it sends using the `–webDriverLogDir` option.

> **INFO** The WebDriver protocol is actually a W3C specification. If you're interested in seeing all the details of the protocol, it's available at https://www.w3.org/TR/webdriver/

This option specifies a directory in which Protractor will create a log of the WebDriver commands it sends during a test run. Each log file will be named `webdriver_log_<sessionId>.txt`, with a different `sessionId` for each test run. Each line of the log file shows when the command was sent, how long it took, and what they command was. It looks like this:

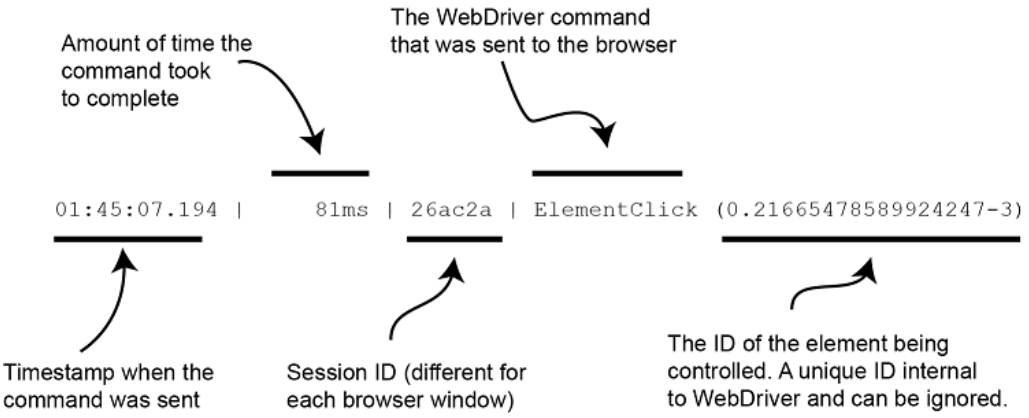

Figure 10.4 The structure of a WebDriver log line

Let's run our add contact test with WebDriver logging. Unlike the other examples thus far, logging WebDriver commands requires running a Selenium server (we'll explain why in a bit, don't worry). To run the test in `chapter10/test_experimental` and generate a WebDriver log, first start a selenium server:

```
webdriver-manager start
```

Then, in another terminal, start the test like so:

```
protractor --webDriverManager .
```

After the test runs, Protractor will create a log of WebDriver commands in the current directory. The entire log is a bit long, but here's a sample of the end of the log after running the example test:

Listing 10.14 Excerpt from WebDriver log

```
01:45:06.984 |    520ms | 26ac2a | Waiting for Angular                          (1)
01:45:06.984 |     14ms | 26ac2a | FindElements                                 (1)
    Using css selector '*[id="contact-name"]'
    Elements: 0.21665478589924247-2
01:45:07.009 |      9ms | 26ac2a | Waiting for Angular                          (1)
01:45:07.009 |    131ms | 26ac2a | ElementSendKeys (0.21665478589924247-2)
    Send: Ada Contact                                                           (1)
01:45:07.165 |     23ms | 26ac2a | Waiting for Angular
01:45:07.166 |     18ms | 26ac2a | FindElements                                 (2)
    Using css selector '.create-button'                                         (2)
    Elements: 0.21665478589924247-3                                             (2)
01:45:07.194 |      8ms | 26ac2a | Waiting for Angular
01:45:07.194 |     81ms | 26ac2a | ElementClick (0.21665478589924247-3)
01:45:08.306 |   1028ms | 26ac2a | Waiting for Angular                          (3)
01:45:08.306 |     15ms | 26ac2a | FindElements                                 (4)
    Using css selector 'app-contact-list'                                       (4)
    Elements: 0.21665478589924247-4                                             (4)
01:45:08.331 |      9ms | 26ac2a | Waiting for Angular                          (4)
01:45:08.331 |     47ms | 26ac2a | GetElementText (0.21665478589924247-4 )      (4)
    Name Email Number
mood Adrian Directive adrian.directive@example.com +1 (703) 555-0123 edit delete
mood Rusty Component rusty.component@example.com +1 (441) 555-0122 edit delete
mood Jeff Pipe jeff.pipe@example.com +1 (714) 555-0111 edit delete
mood Craig Service craig.services@example.com +1 (514) 555-0132 edit delete
mood Ada Contact edit delete

Delete All Contacts
add
01:45:08.388 |     54ms | 26ac2a | DeleteSession
```

(1) Typing 'Ada Contact' into the contact name field.
(2) Finding and clicking the 'Create contact'; button.
(3) After clicking the create button, Protractor waits about a second for the contact list to load.
(4) Getting the text of the contact list.

You might have noticed that all the "Waiting for Angular" log lines. What do these mean? Remember from Chapter 9 that Protractor waits for Angular to be stable before sending a WebDriver command – every time it does so, it adds "Waiting for Angular" to the log. The log also shows the amount of time that each step takes to complete – you can see that after clicking the "Create button", Protractor waits for Angular for 1028ms while the contact list loads.

The WebDriver log is handy for knowing what happened in a test after the fact, but sometimes it would be nice to be able to just watch the browser while the test runs. Usually, the test runs too quickly to be able to tell what's happening, but you can slow it down using Highlight Delay.

10.3.2 Highlight Delay

When Protractor tests run, they execute WebDriver commands as fast as possible. If you've ever tried to debug a Protractor test by watching the browser window, you've seen

what this looks like. Buttons are clicked so fast, it's as if the browser is being controlled by a hyperactive poltergeist. It's normally impossible to follow what's happening.

We can fix this with the `–highlightDelay` flag. This tells Protractor to add a delay, specified in milliseconds, before sending WebDriver commands to the browser. Protractor will also highlight the element it's about to touch with a light blue rectangle, so we can tell which element is about to be clicked. For example, if the test is about to enter text in a field, it'll first highlight the text field with a blue rectangle, and then wait the specified delay before proceeding.

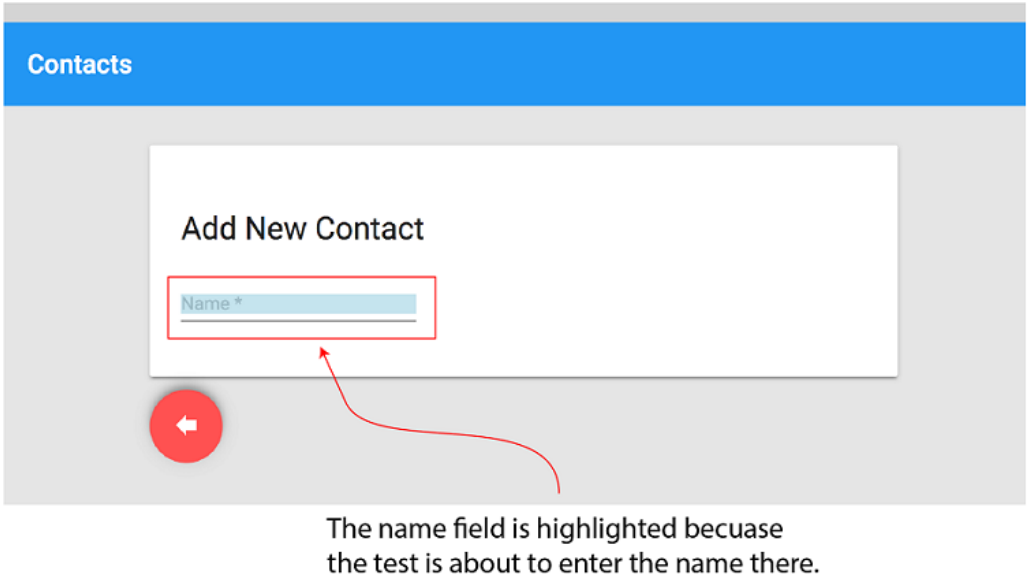

Figure 10.5 The name input field is highlighted before a name is entered.

Adding a Highlight Delay can be a quick and easy way to see what's going on during a Protractor test. You can slow things down as much you like, and the highlight lets you know which part of the page is going to be touched next.

10.3.3 Blocking Proxy

Both Highlight Delay and WebDriver Logs are experimental new features that are made possible by a new component in Protractor called Blocking Proxy. This is a proxy that sits between your test and the browser driver. This proxy can intercept and optionally delay any command that your test sends to the browser. Thus, it can create a log of commands, or delay commands.

One interesting benefit of having this functionality in a proxy is that it can be used with any WebDriver test. This means you don't have to use Protractor in order to use it – even if you have WebDriver tests written in Java or Python, you could still use Blocking Proxy to add Highlight Delay to your tests. It even implements the same waiting for Angular logic that Protractor uses. This means that if you don't feel like writing your tests in TypeScript and running them on NodeJS, you could write them in any language that WebDriver supports and use Blocking Proxy to get the same "wait for Angular" behavior as Protractor.

Blocking Proxy is still an experimental feature in Protractor. If you'd like to learn more about it or use it in other projects, check it out at https://github.com/angular/blocking-proxy.

10.4 Debugging with Chrome Devtools

Debugging a failing Protractor test can be frustrating, especially because there's no obvious way to step through the test and see what it's doing. Fortunately, recent changes to WebDriver, Node.js, and Protractor can make debugging Protractor tests as easy as debugging any traditional application. To better understand the significance of these changes, and why Protractor tests are difficult to debug, we need to understand asynchronous programming in JavaScript, and specifically how the async/await feature added in ES2017 makes asynchronous programming easier.

If you're already familiar with asynchronous programming in JavaScript, feel free to skip to the last section, where you'll see how easy it is to step through Protractor tests using Chrome devtools.

10.4.1 Asynchronous functions and promises

In a language with threads, like Java or C#, we can write code that will block until an I/O operation (that is, reading from the filesystem, sending data over the network, and so on) finishes. However, in JavaScript, when we have an I/O operation, we typically pass a callback that will be invoked when the operation finishes.

Let's imagine a simple function that makes two API calls and writes some result to disk based on the responses. In Java, our imaginary function might look something like the following:

```
responseA = callServiceA();
responseB = callServiceB(responseA);
resultFile = writeResponseData(responseB);
doSomethingWithResult(resultFile);
```

Each function blocks until it finishes, running line by line in an *imperative* (or some might say *synchronous* manner). In JavaScript, we can't make blocking calls; instead, we pass a callback that's invoked when the function is done. The preceding imaginary example might look like the following in JavaScript:

```
callServiceA((responseA) => {
  callServiceB(responseA, (responseB) => {
```

```
    writeResponse(responseB, (resultFile) => {
      doSomethingWithResult(resultFile);
    })
  })
})
```

Depending on when your callbacks are invoked, your code may not execute the same way it reads. Asynchronous programming is kind of like being a time traveler: things don't necessarily happen in the order you expect. This can be confusing for people learning about callbacks for the first time.

For the code in listing 10.17, the first line that prints something to the console is the last line in the listing. The order in which the callbacks are invoked and printed to the console depends on when the different `setTimeout` calls are scheduled, and may not necessarily run in the same order you see when reading the code from top to bottom.

Listing 10.15 Asynchronous program flow example

```
setTimeout(() => {
  console.log('Event C');  ❸  After 1s, this timer filers
},1000);

setTimeout(() => {
  console.log('Event B');  ❷
  setTimeout(() => {
    console.log('Event D'); ❹
  },1000);
},500);

console.log('Event A');  ❶
```

❶ This is executed first
❷ After 500ms, this callback runs and sets another timer
❸ After 1s, this timer fires
❹ A second after Event B happens, this timer fires

Having to pass a callback every time we want to do something with the result of an asynchronous call can lead to the "pyramid of doom" – that is to say, if we have a long list of nested callbacks, we can end up with deeply-indented code that's hard to read. One fix for this is to use promises. Instead of passing a callback to be invoked when the operation is done, we can return a promise that eventually will resolve to the result of the operation. The result is much easier to read:

```
callServiceA().then((responseA) => {
  return callServiceB(responseA)
}).then((responseB) => {
  return writeResponse(responseB)
}).then((resultFile) => {
  doSomethingWithResult(resultFile);
});
```

The WebDriver commands in Protractor tests are asynchronous calls; our test is making an API call to Selenium to send the command to the browser. However, we can normally write Protractor tests without worrying about callbacks and promises thanks to a helpful feature of WebDriver called the Control Flow.

10.4.2 WebDriver Control Flow

Most of the tests we've written so far seem to be synchronous, even though they execute WebDriver commands. How is this possible? The trick is that WebDriver commands return a special kind of promise (called a *managed* promise) that's executed later. Unlike a normal promise, a managed promise doesn't actually run an asynchronous task. Instead, it schedules a command to run on Webdriver's *control flow*. At the end of your test, the control flow runs, and all the scheduled browser commands execute. Thus, you can write your tests as if you were writing asynchronous code without worrying about promises. Listing 10.18 shows how you might write a test relying on managed promises.

10.16 Test using Control Flow

```
it('should open the dialog with waitForAngular', () => {
  let feedButton = element(by.css('button.feed-button'));
  let closeButton = element(by.css('button[mat-dialog-close]'));
  let dialogTitle = element(by.css('app-contact-feed h2.mat-dialog-title'))

  feedButton.click();  (1)
  expect(dialogTitle.getText()).toContain('Latest posts from Craig Service')
  debugger;  (2)

  closeButton.click(); #c
  browser.wait(EC.stalenessOf(dialogTitle), 3000, 'Waiting for dialog to close');
  expect(dialogTitle.isPresent()).toBeFalsy();
}); (4)
```

(1) This doesn't immediately send a click command – it schedules one on the Control Flow.
(2) If we drop into the debugger here, none of the commands have executed yet.
(3) The same is true here – when this test is executed, the browser isn't actually doing anything.
(4) The test commands don't actually run until the it() block is complete.

Unfortunately, the using the control flow prevents us from debugging Protractor tests with standard Node.js tools. If you've ever used the `debugger` keyword in a Protractor test, you've seen this. When the debugger hits the breakpoint, the browser isn't actually executing commands; instead, the test synchronously defines a list of commands to run (that is, the Control Flow). Protractor automatically runs those commands after each `it()` block, which is why you don't see the commands running if you set a breakpoint in your test.

You don't have to use the control flow. You can also treat managed promises as if they were regular promises. You can schedule callbacks to run when the command is executed using `.then()`, as in listing 10.19.

Listing 10.17 Explicitly using WebDriver promises

```
it('should open the dialog with waitForAngular', (done) => {
  let feedButton = element(by.css('button.feed-button'));
  let closeButton = element(by.css('button[mat-dialog-close]'))
  let dialogTitle = element(by.css('app-contact-feed h2.mat-dialog-title'))

  return feedButton.click().then(() => {  ❶
    return dialogTitle.getText();    ❷
  }).then((dialogText) => {
    expect(dialogText).toContain('Latest posts from Craig Service')
    return closeButton.click();
  }).then(() => {
    return browser.wait(EC.stalenessOf(dialogTitle), 3000, 'Waiting for dialog to close');
  }).then(() => {
    return dialogTitle.isPresent();  ❸
  }).then((dialogTitleIsPresent) => {
    expect(dialogTitleIsPresent).toBeFalsy();
    done();   ❹
  });
```

❶ We can wait on the promise returned by click() to wait for the command to finish.
❷ Getting text is also an asynchronous command, so we need to wait on the promise.
❸ Reading anything from the browser about the state of the page, even if an element is present, requires waiting on a promise
❹ Because our test is now asynchronous, we need to call done() when it's finished.

Even though the tests in listings 10.18 and 10.19 do the same thing, the test that uses the control flow (in listing 10.18) is much easier to read. This is why the authors of Selenium WebDriver added the control flow. However, the new async/await feature coming to JavaScript (and available in Node v8) makes writing asynchronous code much more readable than using promises. That's why the Selenium team has decided to deprecate the control flow (see GitHub issue: https://github.com/SeleniumHQ/selenium/issues/2969), and why the control flow will not be used in Selenium 4.x and greater. When using this version of Selenium, you'll no longer be able to write tests as in listing 10.18. Instead, you'll need to explicitly write asynchronous tests, but without the Control Flow your tests will also be much easier to debug.

10.4.3 The future: async/await

The async and await keywords are a new addition to JavaScript that makes asynchronous code much more readable. The details of async/await are beyond the scope of this book, but it's basically syntactic sugar that makes waiting on promises easier.

We can disable the WebDriver Control Flow by adding `SELENIUM_PROMISE_MANAGER: false` to our Protractor config. As of Selenium 4.x, running without the Control Flow will be the only option, but setting this flag gives us a way to get our tests ready for Control Flow deprecation. Using async/await, we can rewrite the test from the following example (listing 10.20) to be much more readable, even though it doesn't use the Control Flow.

Listing 10.18 Test using async/await

```
it('should open the dialog with waitForAngular', async () => { ❶
  let feedButton = element(by.css('button.feed-button'));
  let closeButton = element(by.css('button[mat-dialog-close]'))
  let dialogTitle = element(by.css('app-contact-feed h2.mat-dialog-title'))

  await feedButton.click();
  let dialogText = await dialogTitle.getText(); ❷
  expect(dialogText).toContain('Latest posts from Craig Service')
  debugger ❸

  await closeButton.click();
  await browser.wait(EC.stalenessOf(dialogTitle), 3000, 'Waiting for dialog to close');
  let dialogTitleIsPresent = await dialogTitle.isPresent();
  expect(dialogTitleIsPresent).toBeFalsy(); ❹
});
```

❶ We can only use await in functions that have been declared async.
❷ Instead of having to use .then(), we can use await to get the result of the promise.
❸ With the control flow disabled, now we can start the debugger before the close button is clicked.
❹ Because our function is declared async, we don't need to worry about calling done() when our test is finished.

Notice how using async/await makes our test much more readable, compared with waiting on promises using `.then()`. This is why the Selenium team feels comfortable removing the Control Flow in Selenium 4.x; JavaScript has finally evolved to the point where writing asynchronous code is easy. Disabling the Control Flow lets us do something that hasn't been possible until now – debugging our tests using the Chrome Inspector!

10.4.4 Using Chrome DevTools

Node.js v6 added the ability to debug Node.js programs using Chrome DevTools. The test in listing 10.20 has a debugger statement to set a breakpoint right before the close button is clicked. We can debug our test by starting Protractor with the following command:

```
node --inspect --debug-brk \
    ./node_modules/protractor/bin/protractor  ./debugging_test.conf.js
```

> **REMEMBER** WebDriver Control Flow needs to be disabled by adding `SELENIUM_PROMISE_MANAGER: false` to your Protractor config.

If we start Protractor with Node.js debugging enabled, we'll see something like this:

```
$ node --inspect --debug-brk \
>      ./node_modules/protractor/bin/protractor  ./debugging_test.conf.js
Debugger listening on port 9229.
Warning: This is an experimental feature and could change at any time.
To start debugging, open the following URL in Chrome:
    chrome-devtools://devtools/remote/serve_file/@60cd6e859b9f557d2312f5bf532f6a
ec5f284980/inspector.html?experiments=true&v8only=true&ws=127.0.0.1:9229/362a062
c-9ab4-494b-8331-f8dd684008e1
Debugger attached.
```

Figure 10.6 Starting Protractor with debugging enabled

In Chrome 60 or later, you can debug Node.js programs by opening `about://inspect`. Any Node.js programs ready for debugging l automatically appears in a list.

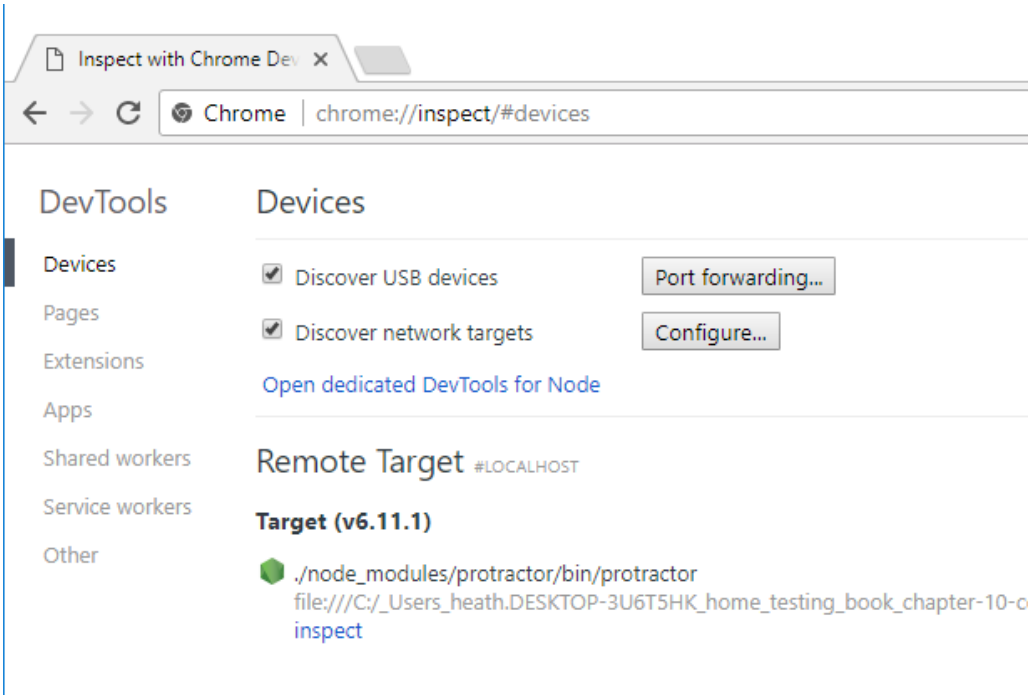

Figure 10.7 Opening DevTools in Chrome

If we select "Open dedicated DevTools for Node", we'll get a debugger attached to our Protractor test. We can step through to the breakpoint we set with the `debugger` statement.

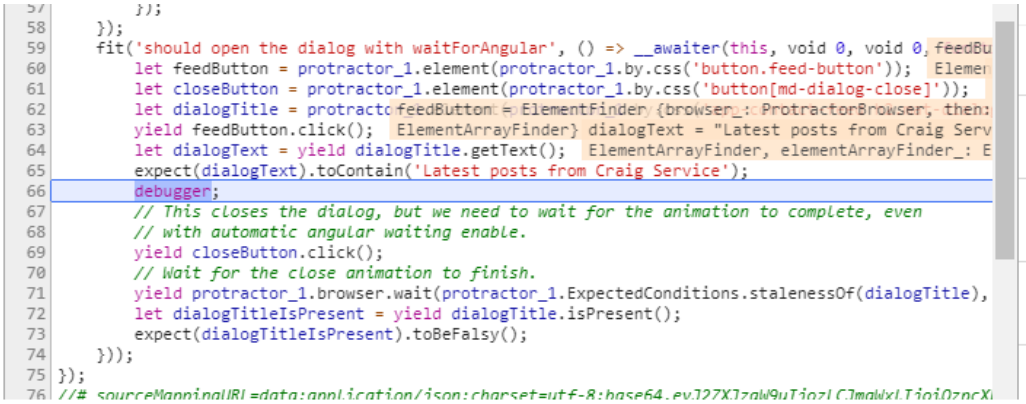

Figure 10.8 Debugging a Protractor test with the Chrome Inspector

If you have some experience writing Protractor tests, you might have seen old debugging tools like ElementExplorer or `browser.pause()`. These tools were necessary when WebDriver's Control Flow prevented debugging a Protractor test like a normal Node.js program. However, now that the Control Flow is deprecated, we can use the much more powerful Chrome DevTools. The old tools should not be used.

10.5 Summary

Protractor is a powerful tool, and this chapter is intended to inspire some new ways to use Protractor in your own projects. In this chapter, you learned the following:

- You can pass command line flags to the browser using the capabilities section of your Protractor config. For example, you can pass `'--headless'` to Chrome to start it in headless mode.
- Plugins let you add features to Protractor, like changing how test results are reported or taking screenshots on test failure.
- Protractor can take screenshots of your application during a test, and you can use this to create tests that verify your application web interface looks right.
- The WebDriver Control Flow is deprecated and will not be in Selenium 4.x, and you should start using async/await in your tests.
- It's easy to debug tests using Chrome DevTools.

11

Continuous integration

Writing tests for your Angular application is only half the battle. You need to remember to run them as well, so that you catch regressions as you continue to add features. Running tests manually can be tedious. It's better to set up a continuous integration (CI) system that will integrate with our source repository and automatically run all the tests for each change.

In practice, you develop your Protractor tests on a desktop, where you can watch the browser and see if your tests are passing. One big stumbling block people run into when setting up their tests in a CI system is that the server doesn't normally have a GUI (*headless environment*). In this chapter, we'll show you how to set up your tests to run automatically in a headless environment. First, we'll cover setting up your own CI using the open source Jenkins server, then we'll show you how easy it can be to set up testing with Circle CI, which is a hosted CI service that has a free tier.

11.1 Jenkins

Jenkins is an open-source continuous integration server with a powerful web interface. It was originally created for testing Java applications, but thanks to a rich ecosystem of plugins, Jenkins now can test almost any project in any language. Let's set up a Jenkins server that will run our tests on NodeJS.

11.1.1 Setting up Jenkins

Jenkins is a large, complicated project, and the full details of installing it are out of scope of this book. The example in this chapter assumes you're running Jenkins on an Ubuntu server, but you can also install it on OSX and Windows. Follow the official instructions for installing Jenkins on your server at https://jenkins.io/doc/book/installing and then come back here when you've set up the first administrator user.

Both our unit and E2E tests need a browser to run; we'll use Chrome for this example. Chrome needs a GUI to run and our server is headless, so we'll install Xvfb. Of course, we also need NodeJS to run the tests.

> **INFO** On Linux, the GUI is provided by an *X Server*. Xvfb is an X server that uses a *virtual framebuffer* for display. This means that the graphical display is entirely in memory. Xvfb is a good tool to use if we need to use graphical programs on a Linux server that may not have a display attached.

Listing 11.1 shows how to install the prerequisites on Ubuntu. We can use apt to install Xvfb, but we manually install Google Chrome and Node Version Manager (nvm) to get the latest versions of each.

Listing 11.1 Install Prerequisites: Chrome, Xvfb, and Node

```
sudo apt-get update
sudo apt-get install Xvfb -y                                          (1)

sudo sh -c 'echo "deb [arch=amd64] \
    http://dl-ssl.google.com/linux/chrome/deb/ stable main" >> \
    /etc/apt/sources.list.d/google-chrome.list'                       (2)

wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub \
    | sudo apt-key add -

sudo apt-get update
sudo apt-get install google-chrome-stable –y

curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.33.2/install.sh \
    | bash                                                            (3)
nvm install 8                                                         (4)
```

(1) Install Xvfb
(2) Install Google Chrome
(3) Install a node version manager
(4) Use node version manager to install node version 8

Now that we have our prerequisites, we're ready to install the plugins we need using Jenkin's browser-based admin interface. In the browser window, you can install plugins by navigating to Manage Jenkins and then clicking the Manage Plugins link. The plugins we need for our Jenkins job are the JUnit plugin, the Xvfb plugin, and the nvm wrapper plugin, all shown in figure 11.1. The JUnit plugin will interpret the test and show a trending history of your test. The Xvfb and nvm plugin will allow your test to launch Protractor with NodeJS and run your test in a browser.

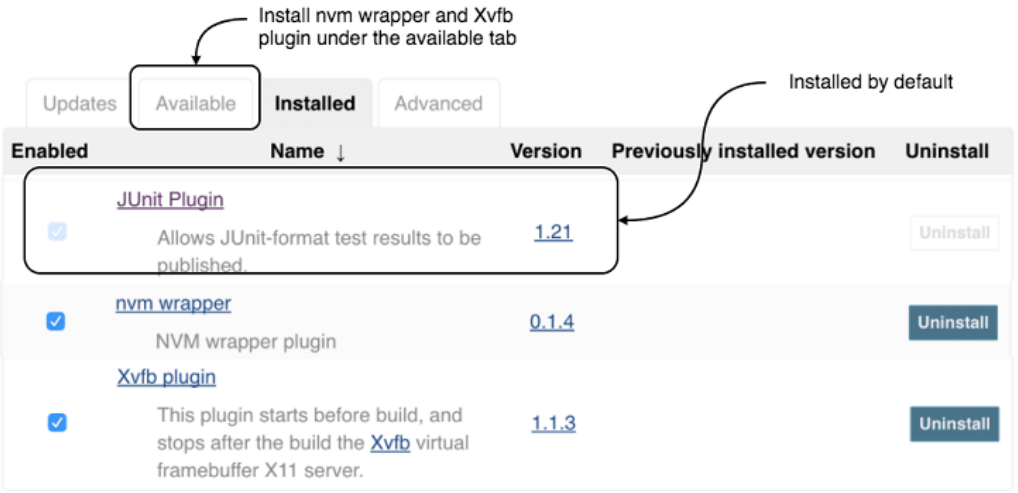

Figure 11.1 JUnit, nvm wrapper, and Xvfb plugins

We'll also need to set some configuration options for Jenkins. In the same browser window, you can configure Jenkins by clicking Manage Jenkins, and then navigate to Global Tool Configuration. From there, you need to set a valid path for both JAVA_HOME and Xvfb, shown in figure 11.2. Jenkins requires Java to be set in the path to launch the selenium standalone server and Xvfb to run the virtual desktop environment.

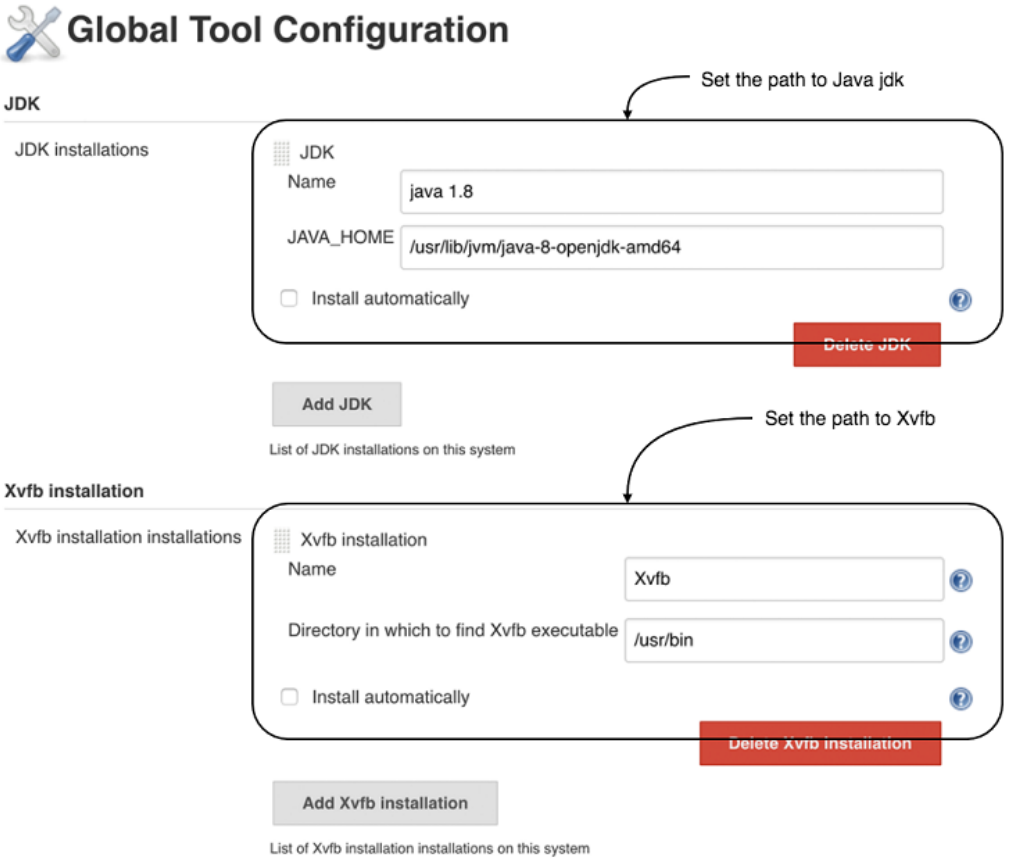

Figure 11.2 Jenkins Global Tool Configuration

Now that you have all the components required to run a Protractor test, let's review the test in the GitHub repository (`https://github.com/testing-angular-applications/testing-angular-applications/`). In Jenkins, create a new freestyle project and add the git repository as https://github.com/testing-angular-applications/testing-angular-applications.git in Source Code Management.

11.1.2 Unit tests

We use Karma to run our unit tests. Karma takes care of starting a browser, connecting to it, and running your tests there. Because Karma runs our unit tests in a real browser, our tests have access to a real DOM and our client code runs exactly as it will in production. In fact, you can use Karma to run your unit tests on other browsers like Firefox and Safari, to make sure

your application is compatible with them. To keep things simple, for this example we'll just run the unit tests for our project on Chrome.

Jenkins expects test results to be in an XML file with the same format the JUnit would use to report Java test results. This format has become a de-facto standard, and now test frameworks in a variety of languages can report test results in this JUnit-XML style. Karma has a plugin called karma-junit-reporter, which outputs the results of running Karma in this format so that Jenkins can understand them. To run our unit tests on Jenkins, we'll create a separate configuration for Karma that uses this plugin.

Chrome recently added *headless mode*. This starts Chrome without a display. Our unit tests don't need a GUI, so even though we've already set up Xvfb, let's go ahead and use headless Chrome to run our unit tests. The karma-chrome-launcher plugin will start headless Chrome if you specify the browser as ChromeHeadless. Listing 11.2 shows the Karma config for our CI server, using headless Chrome and JUnit XML reporting.

Listing 11.2 chapter11/karma-ci.conf.js

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular/cli'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'),
      require('karma-junit-reporter'),
      require('@angular/cli/plugins/karma')
    ],
    reporters: ['junit'],                    (1)
    junitReporter: {
        outputDir: 'karma-results',          (2)
        outputFile: 'karma-results.xml'      (2)
    },
    angularCli: {
      environment: 'dev',
    },
    port: 9876,
    logLevel: config.LOG_INFO,
    browsers: ['ChromeHeadless'],            (3)
       autoWatch: false,                     (4)
    singleRun: true                          (4)
  });
};
```

(1) Use the JUnit reporter, to create a JUnit-style XML.
(2) Output results in the 'karma-results' directory
(3) Run the tests in headless Chrome
(4) In CI, run the tests only once and don't watch for changes

This is a Karma configuration separate from the one we use during development. For the rest of this book, we've kept the test cases and website in separate directories, so you can follow along as tests are written in each chapter.

NOTE In practice, it's best to have unit tests next to the code they test. The `chapter11/run_unit_ci.sh` script in the project repo (https://github.com/testing-angular-applications/testing-angular-applications) copies the tests and Karma config to the `website/` directory before running Karma.

Create a new project in Jenkins by clicking New Item on the main page. Create a new Freestyle project. In the configuration for the new project, select Git under Source Code Management. The build config for our unit tests is shown in figure 11.3.

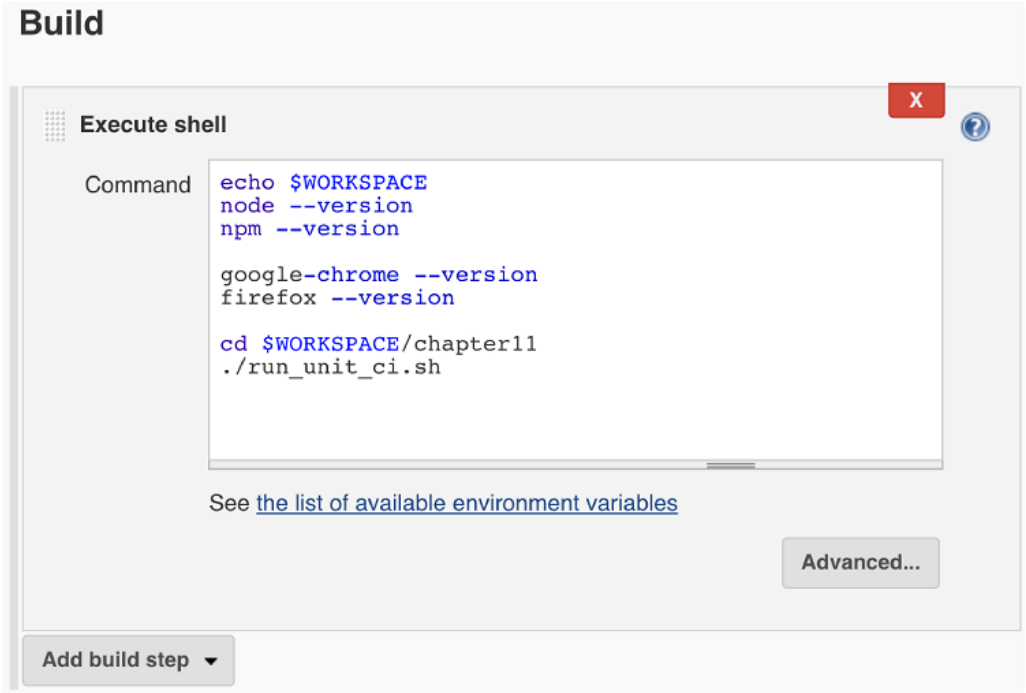

Figure 11.3 Build configuration for the unit test Jenkins job

All this config needs to do is call the `run_unit_ci.sh` script. In general, it's better to keep as much of your CI process as possible in a script in your source repo, where it's easy to debug and track changes. This also keeps your Jenkins config simple, because all it needs to do is call that script. Our Jenkins job config also prints out the versions of npm, Node, and Chrome. Jenkins saves the console output for each run, and it can be helpful to see exactly which versions of these tools was used in an old run.

We also need to tell Jenkins where to find the JUnit XML output for our test run. That configuration is done in the Post-build Actions section, as shown in figure 11.4.

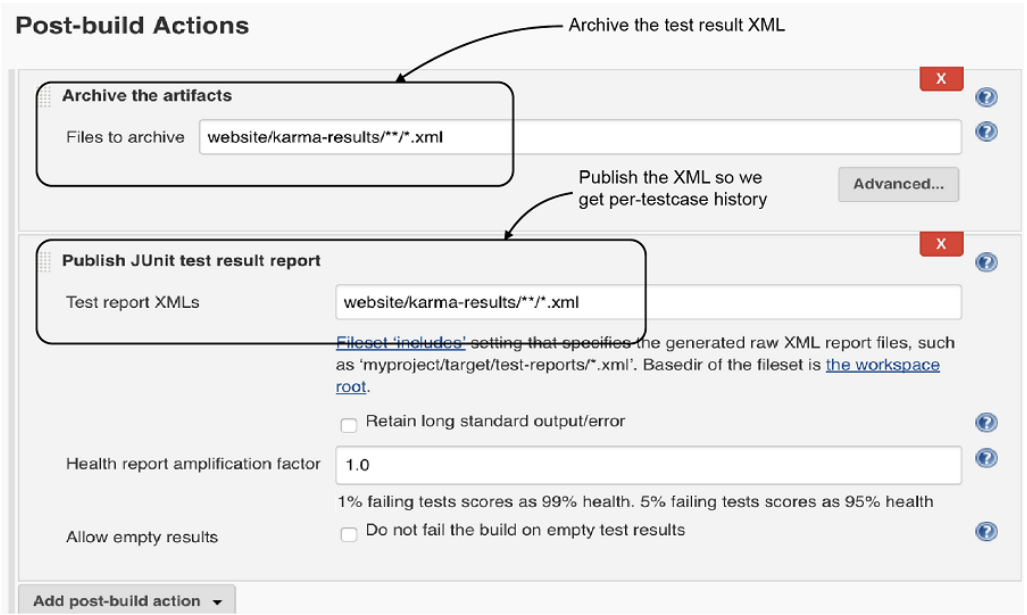

Figure 11.4 Post-build actions for the unit test Jenkins job

Note that we both archive the XML and publish a JUnit test report based on it. Publishing the JUnit test report will create a chart of test pass/failure over time, and it also give us detailed information about which test cases failed. Archiving the XML means that Jenkins will keep the file around, which can be helpful later for debugging purposes.

11.1.3 E2E tests

Setting up Jenkins to run E2E tests is quite similar to setting it up to run unit tests. We could use headless Chrome again, but for instructional purposes, let's try something different and use regular Chrome and Xvfb. Also, instead of using directConnect as we've done in previous chapters, we'll use a local driver provider, which will tell Protractor to launch the selenium server on a random available port. We use this option because we might not have a selenium standalone server running in the background. As before, we will turn on JUnit-style reports so we get detailed information about test runs in Jenkins.

Before we set up Jenkins, let's look at the Protractor configuration file (listing 11.3). We want to set up our tests to behave differently when running on Jenkins. In the configuration file, we set an `IS_JENKINS` environment flag. When our tests aren't running on Jenkins and that flag isn't set, the default behavior will be to use directConnect. In our Jenkins environment, we'll set the `IS_JENKINS` environment variable to true. When this flag is set, the test will not launch with directConnect, and Protractor will handle starting and stopping your

selenium standalone server on your behalf. In addition, the `IS_JENKINS` flag also guards the `onPrepare` function to generate only JUnit reports on Jenkins.

Listing 11.3 chapter11/protractor.conf.js

```
exports.config = {
  directConnect: !process.env.IS_JENKINS,                          (1)
  capabilities: {
    browserName: 'chrome'
  },
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  specs: ['e2e/**/*.spec.ts'],
  onPrepare: () => {
    if (process.env.IS_JENKINS) {                                  (2)
      let jasmineReporters = require('jasmine-reporters');
      let junitReporter = new jasmineReporters.JUnitXmlReporter({
        savePath: 'output/',
        consolidateAll: false
      });
      jasmine.getEnv().addReporter(junitReporter);
    }
    require('ts-node').register({
      project: 'e2e'
    });
  },
  useAllAngular2AppRoots: true
};
```

(1) When IS_JENKINS is set to true, we will not use directConnect and launch with a local driver provider
(2) When IS_JENKINS is set to true, we will create JUnit style reports

When we configure the Jenkins job for our E2E tests, we'll need to first set up the build environment. In the Build Environment section shown in figure 11.5, select the option to Start Xvfb before the build and also to run the build in an nvm-managed environment with Node v8.

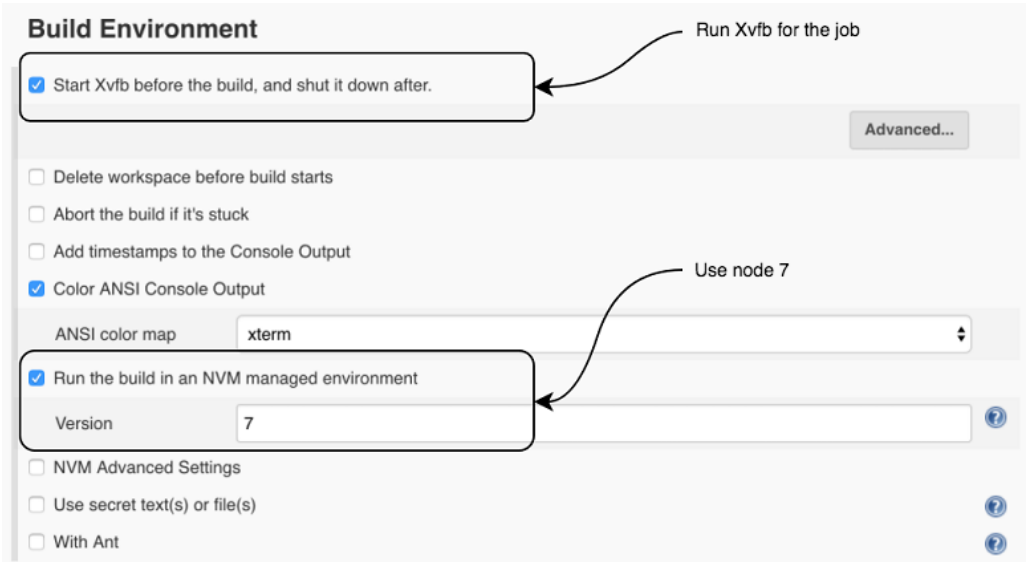

Figure 11.5 Build environment for the E2E Jenkins Job

Next, we set up an Execute shell build step with the snippet shown in listing 11.4. In the shell commands, we specify the environment variable that our Protractor configuration file is using. We defined this previously to not use `directConnect` and to generate JUnit style reports when `IS_JENKINS` is set to true.

Listing 11.4 Jenkins Execute shell

```
export IS_JENKINS=true                    (1)

node --version                            (2)
npm --version                             (2)
google-chrome --version                   (2)

cd test_jenkins                           (3)
npm install                               (4)
npm run e2e protractor.conf.js            (5)
```

1. Set an environment variable
2. Check versions for node, npm and the Chrome browser
3. Change directory to the test_jenkins folder
4. Install node modules
5. Run the protractor test

Finally, in the Post Build Actions section, we Archive the Artifacts for the JUnit reports. We will also Publish JUnit test result report with the same file set we are planning to archive (figure 11.6).

Figure 11.6 Post-build actions for the E2E Jenkins job

Publishing a JUnit test result report will create a nice history in Jenkins of our test runs. After more than one build occurs, the Jenkins job will display a plot of failed and passing tests along with the set of last successful artifact outputs. Figure 11.7, shows the summary plot and artifacts for our test job.

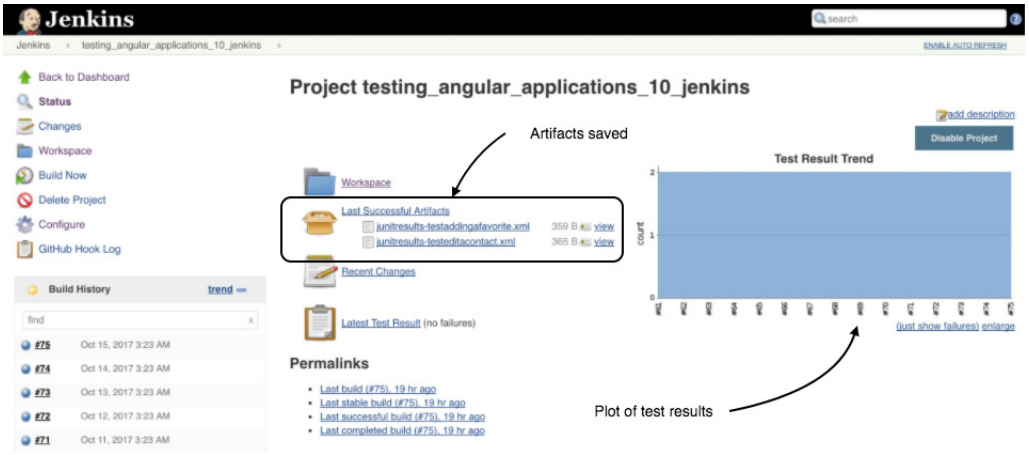

Figure 11.7 Jenkins test results display

As previously mentioned, we could have used headless Chrome instead of Xvfb, and we also could have used directConnect instead of launching the test with the local driver provider option. In the next section, we take the same test and run it on Circle CI, but with headless Chrome and directConnect instead.

11.2 Circle CI

CircleCI is another continuous-integration service that integrates with GitHub and is free for open-source projects; it also has a paid enterprise tier. The neat thing about CircleCI version 2.0 is that it is optimized for docker images. Docker is a container-based system for Linux virtualization, so when we use a docker image, it is a virtual Linux distribution running pre-installed software. We can use a container to install NodeJS and Chrome in our continuous integration environment.

CircleCI makes it easy to run our tests more often. This gives us the option of testing things automatically that might have been tested manually before. For example, we can use our new Protractor CI setup to automatically test the appearance of our app using tests that take screenshots.

Before we set up our CircleCI configuration, let's take our previous Jenkins example and add the environment variable IS_CIRCLE. This additional environment variable will run our CircleCI tests with headless Chrome using `directConnect`. On Jenkins, it will run with a local launch of the selenium standalone server with JUnit-style reports. Finally, in our development environment, we use `directConnect` without headless mode so that we can see what is happening in our E2E tests.

Listing 11.5 Protractor configuration for Circle CI

```
exports.config = {
  directConnect: (!process.env.IS_JENKINS && true),
  capabilities: {
    browserName: 'chrome',
    chromeOptions: {
      args: (process.env.IS_CIRCLE ? ['--headless'] : [])                ❶
    }
  },
  baseUrl: 'https://contacts-app-starter.firebaseapp.com',
  specs: ['e2e/**/*.spec.ts'],
  onPrepare: () => {
    if (process.env.IS_JENKINS) {
      let jasmineReporters = require('jasmine-reporters');
      let junitReporter = new jasmineReporters.JUnitXmlReporter({
        savePath: 'output/',
        consolidateAll: false
      });
      jasmine.getEnv().addReporter(junitReporter);
    }
    require('ts-node').register({
      project: 'e2e'
    });

  },
  useAllAngular2AppRoots: true
};
```

❶ When the environment variable IS_CIRCLE is set to true, use headless Chrome

Next, we need to set up CircleCI by creating a configuration file, `./circle/config.yml`. We can use a Circle CI-maintained docker image that includes Node and browsers. Next, we need to specify the shell commands to run our tests. The only difference in the shell commands between listing 10.12 and the Jenkins shell commands is that in Circle CI, we need to change to the `website` directory and run the npm command.

Listing 11.6 CircleCI configuration file .circle/config.yml

```
version: 2                                                               ❶
jobs:
  build:
    working_directory: ~/workspace                                       ❷
    docker:
      - image: circleci/node:7-browsers                                  ❸
    steps:
      - checkout

      - run: export IS_CIRCLE=true                                       ❹
      - run: node --version
      - run: npm --version
      - run: google-chrome --version

      - run: cd website && npm install
      - run: cd website && npm run karma start karama-ci.conf.js         ❺
```

```
      - run: cd website && npm run e2e protractor.conf.js                6
```

1. Specify to use CircleCI version 2.0
2. Specify the working directory that will be used to checkout the git repository
3. Downloads the CircleCI docker image for Node 7 with browsers
4. Sets the environment variable IS_CIRCLE so our tests will run with headless Chrome
5. Change directory to the website folder then run the unit tests
6. Run the e2e tests

Note that running unit and E2E tests in CircleCI is easy: we just specify a shell command to run. As with Jenkins, CircleCI can take test results in JUnit XML format to give detailed test information. Unlike Jenkins, CircleCI will automatically search your project and try to interpret any test output XML files that it finds. If we don't care about detailed results, we don't have to produce a JUnit XML file; CircleCI will consider the test failed if any of the steps return a non-zero exit code, which Karma and Protractor both do if a test fails.

11.3 Summary

The great part about being software developers is that we have the power to automate tedious tasks. Setting up CI is a great example of this: it automates the task of running all of our tests after each change. As our project and team scales, having CI becomes even more important, because it ensures that all test cases are run with each change, not just those that a developer happened to think of.

In this chapter, you learned about working with CI, including the following:

- How to set up a Jenkins server to run your unit and E2E tests.
- If your tests can produce a JUnit-style XML test result, tools like Protractor and CircleCI can use that information to create detailed reports of your test runs.

CircleCI is an easy-to-use hosted solution for CI with a free tier, and it's good if you need a quick solution for setting up a CI system.

A

Setting up the sample project

This appendix covers:

- Installing prerequisites
- Installing the Angular CLI along with Jasmine, Protractor, and Karma

In this book, we will be using the Angular CLI to run tasks like serving the web application up, execute our tests, and manage our dependencies. In this appendix, we will look at the Angular CLI and install the tool itself. By installing the Angular CLI, you will also install Jasmine, Protractor, and Karma. After we install the Angular CLI, we will get the sample project up and running. Let's dive in!

A.1 Introducing the Angular CLI

One of the major pain points in the past was that setting up an Angular project could be quite challenging and time consuming. In March 2017, the Angular team launched a tool called the Angular Command Line Tool (CLI) to address this very issue. The Angular CLI greatly cuts down the time it takes to setup an Angular project.

At the time of this writing, the Angular CLI contains around a dozen commands that are useful when creating and maintaining Angular applications. In this book, we will mainly utilize only a small number of those commands for testing purposes. Below is a table of the commands that we will be using and their respective description that we will be using throughout the book.

Table A.1: Angular CLI commands that we will be using

Command	Description
ng serve	Builds the Angular application and starts a web server.
ng test	Runs your unit tests.
ng e2e	Runs your end-to-end tests.

For a full list of commands please visit https://github.com/angular/angular-cli/wiki. If you want to learn more about the CLI in general please visit https://cli.angular.io/.

A.2 Installing prerequisites

Before we install the Angular CLI we need to make sure that we have the proper prerequisites installed. There are two prerequisites for installing the Angular CLI:

- Node.js version 6.9.0 or greater
- npm version 3 or greater

To ensure that you have a version of Node.js greater than 6.9.0 or higher run the following command in your terminal:

```
node –v
```

You should see output similar to the following in the terminal:

```
v8.7.0
```

If you need to update your version of Node.js please visit https://nodejs.org/en/ to install the latest version or at least version 6.9.0. If you would like to choose which version to install you can go to https://nodejs.org/dist/ to select whichever version you would like to install.

> **PROTIP: USING NVM** If you need to switch versions of Node.js frequently because of compatibility issues with different projects, I would highly recommend that you use nvm. nvm is the Node Version Manager that makes maintaining versions of Node.js a snap. You can have multiple versions of Node.js installed and switch between the versions whenever you need to. To install and utilize nvm, visit https://github.com/creationix/nvm.

Now that we have Node.js installed, we need to ensure that you have a version of npm that is greater than 3. Run the following command:

```
npm –v
```

> **NOTE: NPM !== NODE.JS PACKAGE MANAGER** It is commonly thought that npm is an acronym for Node.js Package Manager. This is not the case. The first hint is that npm is not capitalized as is common among acronyms. In fact, npm is a recursive backronym that means "npm is not an acronym."

If npm has been installed correctly then you should see something like this in your terminal:

```
5.4.2
```

If you don't see a version greater than version 3, then try reinstalling Node.js by downloading a fresh copy of Node.js at https://nodejs.org/en/. Since npm ships with Node.js, it should give you the most up-to-date version of npm by just installing Node.js. If that doesn't work try updating npm using npm (that is not a typo) by running the following command:

```
npm install npm@latest -g
```

That's it in terms of prerequisites. We can now install the Angular CLI tool. Follow the appropriate below steps to get the Angular CLI up and running.

A.2.1 Installing the Angular CLI the first time

If you have never installed the Angular CLI, run the following command to install it globally:

```
npm i -g @angular/cli
```

That's it! You are good to go! After it is installed, go to the "Verifying the Installation" section.

A.2.2 Updating an old version of the Angular CLI

If you have an old version of the Angular CLI already installed, there are a couple of steps that you may have to take, depending on the version that is installed. The difference is that after version 1.0.0-beta.28, the name and scope of the project was changed from `angular-cli` to `@angular/cli`. To find out which version you have installed, run the following command:

```
ng –V
```

You should see output of the current version of the Angular CLI that is installed.

UPDATING VERSIONS EQUAL TO 1.0.0-BETA.28 OR LESS

If you have a version installed equal to or lower than 1.0.0-beta.28, run the following commands:

```
npm uninstall -g angular-cli
npm cache clean
npm install -g @angular/cli@latest
```

Then go to the "Verifying the Installation" section.

UPDATING VERSIONS GREATER THAN 1.0.0-BETA.28

If you need to install a version greater than 1.0.0-beta.28, run the following commands:

```
npm uninstall -g @angular/cli
npm cache clean
npm install -g @angular/cli@latest
```

VERIFYING THE INSTALLATION

Now that you have Angular CLI installed, let's make sure that it is installed properly before going any further. Run the following command to make sure that everything has installed properly:

```
ng -v
```

If all goes well, your output should look something like figure A.1.

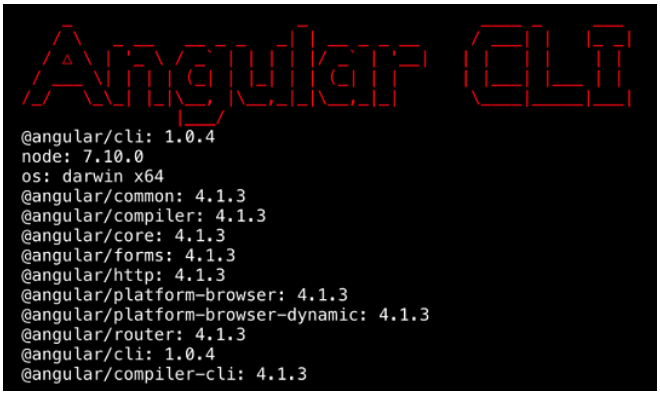

Figure A.1: The Angular CLI version screen

NOTE If you don't see output like figure A.1 or any type of verification saying that the Angular CLI is installed, visit the installation section of the Angular CLI README at https://github.com/angular/angular-cli/blob/master/README.md#installation or the updating section at https://github.com/angular/angular-cli/blob/master/README.md - updating-angular-cli. Otherwise, check out the Testing Angular Applications forum at https://forums.manning.com/forums/testing-angular-applications. There you can post questions or you may find that someone else has run into the same issue and have already have an answer to the issue.

A.3 Installing the sample project

Let's go ahead and clone the sample code for the project. Navigate to a directory in your terminal where you would like to store the sample code for your project. I store my code at ~/Software/GitHub, for example. Then clone the repo by running the following command:

```
git clone https://github.com/testing-angular-applications/testing-angular-applications.git
```

After you've cloned the site, run the following command to navigate to the directory with our app:

```
cd testing-angular-applications/website
```

A.4 Installing dependencies

We need to make sure that we have all the necessary dependencies that the application requires by running the below command:

```
npm install
```

PROTIP ALIASES You are going to do a lot of typing in this book and throughout your career. Knowing aliases will save you valuable time and increase your efficiency. For example, you can use `npm i` instead of `npm install`. In this book, I will introduce aliases so you can practice using them. If you want to learn more about possible shortcuts and aliases, run the following command:

```
npm <command> --help
```

You can also use the `--h` flag (shorthand for `--help`). If viewing the help menu on your console is a little challenging, check out the npm documentation online at https://docs.npmjs.com/ under the "CLI Commands" section.

A.5 Running the application

To run the application run the following command in your terminal:

```
ng serve
```

After running the command (You can also abbreviate "serve" to just "s"), you should see something like the screenshot in figure A.2.

The sample project is now up and running and we are ready to write some tests!

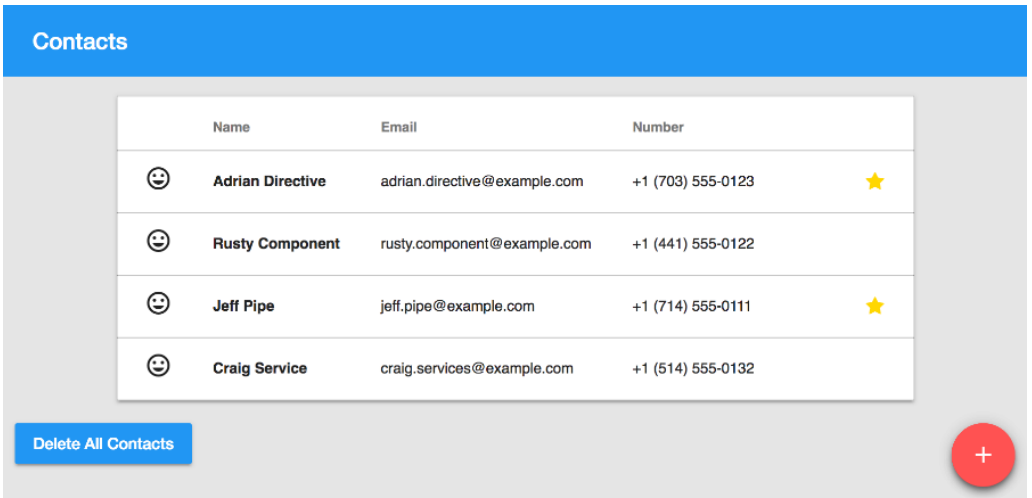

Figure A.2: Contacts App Screen

B

Additional resources

If you are looking for additional testing resources for additional learning, then you are in luck! The following collection of websites, blog posts, videos, and books will help you tofurther sharpen your testing skills.

B.1 Angular testing

Resource	Description	Link
Websites	Angular official testing documentation	https://angular.io/guide/testing
Blog posts	Angular – Testing Guide (v4+), Gerard Sans	https://medium.com/google-developer-experts/angular-2-testing-guide-a485b6cb1ef0
Online course	Testing Angular 4 Apps with Jasmine, Mosh Hamedani	https://www.udemy.com/testing-angular-apps

B.2 General testing

- *The Art of Unit Testing with examples in C#*, Second Edition, Roy Osherove, Manning Publications, 2013.
- *Effective Unit Testing*, Lasse Koskela, Manning Publications, 2013
- *Test Driven*, Lasse Koskela, Manning Publications, 2007
- *Test Driven Development*, Kent Beck, Addison-Wesley Professional, 2002
- *Growing Object-Oriented Software, Guided by Tests*, Steve Freeman and Nat Pryce, Addison-Wesley Professional, 2009
- *Agile Testing: A Practical Guide for Testers and Agile Teams*, Lisa Crispin, Janet Gregory, Addison-Wesley Professional, 2009

- *More Agile Testing: Learning Journeys for the Whole Team*, Lisa Crispin, Janet Gregory, Addison-Wesley Professional, 2014
- *Developer Testing: Building Quality into Software*, Alexander Tarlinder, Addison-Wesley Professional, 2016