Md. Ziaul Haq

# Angular Test-Driven Development

## Second Edition

Enhance your testing skills to build powerful and fault-free applications in Angular v4

Packt>

# Angular Test-Driven Development

*Second Edition*

Enhance your testing skills to build powerful and fault-free applications in Angular v4

**Md. Ziaul Haq**

Packt>

BIRMINGHAM - MUMBAI

# Angular Test-Driven Development

*Second Edition*



First published: January 2015

Second edition: February 2017

Production reference: 1010217

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78646-547-4

www.packtpub.com

# Credits


**Author**

Md. Ziaul Haq

**Reviewers**

Ramachandra Vellanki
Huseyin Babal

**Commissioning Editor**

Wilson D'souza

**Acquisition Editor**

Smeet Thakkar

**Content Development Editor**

Samantha Gonsalves

**Technical Editor**

Huzefa Unwala

**Copy Editor**

Shaila Kusanale

**Project Coordinator**

Devanshi Doshi

**Proofreader**

Safis Editing

**Indexer**

Tejal Daruwale Soni

**Graphics**

Jason Monteiro

**Production Coordinator**

Shraddha Falebhai

# About the Author

**Md. Ziaul Haq** is a senior software engineer, Full Stack JavaScript developer, frontend architect, speaker, and certified scrum master from Dhaka, Bangladesh. He is a passionate programmer who loves to work on web technologies, mobile application development, and new technologies in general. JavaScript is his current technology stack of choice, in which he is working with Angular, NodeJS, ES2015, ReactJS, VueJS, npm, and socket.io. He has also worked on LAMP technologies (`http://www.lamp-technologies.com/`) since an early stage in his career.

Ziaul currently works with the Upwork platform development team as a JavaScript developer. He started his career in 2006 as a software developer and, in his 10-year-long journey, he has worked at different companies in various tech roles. He also worked with UNICEF Myanmar and Bangladesh as a consultant. He just completed his MSc in computer science at United International University, Bangladesh.

Ziaul likes to involve himself in the technology community, where he leads three local JavaScript communities, including AngularJS Bangladesh and TalkJS. He is very active in the local JavaScript community, famously known as `jquerygeek`. You can follow him on Twitter at `@jquerygeek`.

Ziaul likes to watch sports during his free time. His favorite sports are football and cricket. In spite of being a crazy fan of Barca and Messi for football, he never misses a game when his country, Bangladesh, plays any cricket match.

Ziaul lives in Dhaka with his lovely wife, Richi, sweet son, Aarabi, cute daughter, Aafra, and caring parents. He spends time with family and friends when he is not working.

# About the Reviewers

**Ramachandra Vellanki** is a passionate programmer. He has 13 years of programming experience, has worked in different roles, and has experience in building and maintaining large-scale products/applications. He started his career with IBM iSeries and then worked on C++, MFC, .NET, and JavaScript. Currently, he is working on .NET and JavaScript technologies. He enjoys exploring and learning new technologies.

*I would like to thank my parents (Saroja and Ramaiah), wife (Sirisha), and kids (Abhi and Ani) for their love, understanding and constant support. I also would like to thank all my friends and relatives for their continuous encouragement and support throughout my career and life.*

**Huseyin Babal** is an enthusiast Full Stack software engineer and Google Developer Expert who mainly develops web applications using Java, Node.js, and PHP on the backend; Angular, React, and Twitter Bootstrap on the frontend; and Elasticsearch, MongoDB, and Apache Kafka for some research projects. He is the author of *NodeJS in Action* on Udemy, with over 1500 students. He is also interested in DevOps engineering and applies continuous delivery principles to his projects. He writes tutorials about full-stack development on Tuts+ and Java Code Geeks (`https://www.javacodegeeks.com/`) and shares his experiences in public conferences.

He lives in Istanbul with his wonderful wife and son. Apart from the computer world, he likes to spend time with his wife by walking at least an hour every day, visiting different places, watching cartoons, and going on summer holidays.

# www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.


`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

# Customer Feedback

Thank you for purchasing this Packt book. We take our commitment to improving our content and products to meet your needs seriously--that's why your feedback is so valuable. Whatever your feelings about your purchase, please consider leaving a review on this book's Amazon page. Not only will this help us, more importantly it will also help others in the community to make an informed decision about the resources that they invest in to learn. You can also review for us on a regular basis by joining our reviewers' club. **If you're interested in joining, or would like to learn more about the benefits we offer, please contact us**: `customerreviews@packtpub.com`.

# Table of Contents

# Preface

This book will provide the reader with a complete guide to test-driven development (TDD) focused on JavaScript and then dive into an Angular approach. It will provide clear, step-by-step examples to continually reinforce the best practices in TDD. This book will look at both unit testing with Karma and end-to-end testing with Protractor, and not only focus on how to use the tools but understand the reason they were built and why they should be used. Throughout, there will be a focus on when, where, and how to use these tools, constantly reinforcing the principles of the test-driven development life cycle (test, execute, and refactor).

All examples in this book are based on Angular v2 and are compatible with Angular v4.

## What this book covers

`Chapter 1`, *Introduction to Test-Driven Development*, introduces us to the fundamentals of test-driven development by explaining why and how TDD can help in the development process.

`Chapter 2`, *Details of JavaScript Testing*, covers TDD in a JavaScript context. This chapter explores the types of tests that need in JavaScript applications, such as unit testing, behavior testing, integration testing, and E2E testing. This also explains different types of JavaScript testing tools, frameworks, and their uses in the Angular app.

`Chapter 3`, *The Karma Way*, explores the origins of Karma and why it has to be used in any Angular project. By the end of this chapter, readers will not only understand the problem that Karma solves but also will walk through a complete example using Karma.

`Chapter 4`, *End-to-End Testing with Protractor*, takes a look at end-to-end testing applications, through all the layers of an application. This chapter introduced readers to Protractor, the end-to-end testing tool from the Angular team. Then it explained why Protractor was created and how it solve the problems. Finally, it guide the readers step by step how to install, configure, and use Protractor with TDD on an existing Angular project.

`Chapter 5`, *Protractor, a Step Ahead*, goes a bit deeper with Protractor and explores some advanced configurations. And then it explains the test suite debugging process with examples. This also explores some commonly used Protractor APIs with the relevant example.

`Chapter 6`, *The First Step*, provides an introductory walk-through of how to use TDD to build an Angular application with classes, components, and services. This chapter helps readers to begin the TDD journey and see the fundamentals in action. Up to this point, this book stays focused on a foundation of TDD and the tools. And then, by going a step forward it dives into TDD with Angular.

`Chapter 7`, *Flip Flop*, continues to expand our knowledge of applying TDD with Angular features such as routes and navigation to get the result set through our sample Angular application. Besides the Angular features, this also guides the readers how to apply end-to-end tests for those specific features with the help of Protractor.

`Chapter 8`, *Telling the World*, covers more unit testing for the sample Angular application, including routes and navigation. Besides that, this chapter also refactor the existing code to make it more testable. And then it introduces to Angular services and event broadcasting while refactoring the code and introduce MockBackend to test HTTP requests in the service.

# What you need for this book

In this book, we have used **Node Package Manager** (**npm**) as our command tool to run the application and various testing tools. So it's a prerequisite to have npm globally installed on your operating system. To install it, you must have Node.js installed on your operating system.

We will not look at how to install Node.js and npm. There are a lot of resources already available for installing them on any operating system.

# Who this book is for

This book is for developers who have basic experience with Angular but want to understand the wider context of when, why, and how to apply testing techniques and the best practices to create quality clean code. To get the most out of this book, you should have a good understanding of HTML, CSS, and JavaScript, and a basic understanding of Angular with TypeScript.

# Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The next lines of code read the link and assign it to the to the `Calculator` function."

A block of code is set as follows:

```
var calculator = {
   multiply : function(amount1, amount2) {
       return amount1 * amount2;
   }
};
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Runner</title>
  </head>
  <body>
    // ...
    <script src="calculator.js"></script>
  </body>
</html>
```

Any command-line input or output is written as follows:

```
$ npm install protractor
$ npm protractor --version
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In order to download new modules, we will go to **Files** | **Settings** | **Project Name** | **Project Interpreter**".

Warnings or important notes appear in a box like this.


Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

# Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Angular-Test-Driven-Development-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

# Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/AngularTestDrivenDevelopment_ColorImages.pdf`.

# Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

# Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

# Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

# 1
# Introduction to Test-Driven Development

Angular is at the forefront of client-side JavaScript testing. Every Angular tutorial includes an accompanying test, and event test modules are a part of the core Angular package. The Angular team is focused on making testing fundamental to web development.

This chapter introduces you to the fundamentals of **test-driven development** (**TDD**) with Angular, including the following topics:

- An overview of TDD
- The TDD life cycle: test first, make it run, and make it better
- Common testing techniques

## An overview of TDD

TDD is an evolutionary approach to development, where you write a test before you write just enough production code to fulfill that test and its refactoring.

This section will explore the fundamentals of TDD. Let's take a tailor as an example and see how he would apply TDD to his own process.

# Fundamentals of TDD

Get an idea of what to write in your code before you start writing it. This may sound cliched, but this is essentially what TDD gives you. TDD begins by defining expectations, then makes you meet the expectations, and finally forces you to refine the changes after the expectations are met.

Some of the clear benefits of practicing TDD are as follows:

- **No change is small**: Small changes can cause a lot of breaking issues in the entire project. Practicing TDD is the only way that can help, as the test suite will catch the breaking points and save the project after any change, and will thus save lives of developers.
- **Specifically identify the tasks**: A test suite specifically provides a clear vision of the tasks and the step-by-step workflow in order to be successful. Setting up the tests first allows you to focus on only the components that have been defined in the tests.
- **Confidence in refactoring**: Refactoring involves moving, fixing, and changing a project. Tests protect the core logic from refactoring by ensuring that the logic behaves independently of the code structure.
- **Upfront investment, benefits in the future**: Initially, it looks like testing takes extra time, but it actually pays off later when the project becomes bigger, it gives us confidence to extend the features as just running the test will identify the breaking issues, if any.
- **QA resources might be limited**: In most cases, there are some limitations on QA resources as it always takes extra time for everything to be manually checked by the QA team, but writing some test cases and running them successfully will definitely save some QA time.
- **Documentation**: Tests define the expectations that a particular object or function must meet. An expectation acts as a contract and can be used to see how a method should or can be used. This makes the code readable and easier to understand.

# Measuring the success with different eyes

TDD is not just a software development practice–its fundamental principles are shared by other craftsmen as well. One of these craftsmen is a tailor, whose success depends on precise measurements and careful planning.

## Breaking down the steps

Here are the high-level steps that a tailor performs to make a suit:

1. **Testing first**:
    - Determining the measurements for the suit
    - Having the customer determine the style and material they want for their suit
    - Measuring the customer's arms, shoulders, torso, waist, and legs
2. **Making the cuts**:
    - Selecting the fabric based on the desired style
    - Measuring the fabric based on the customer's body shape
    - Cutting the fabric based on the measurements
3. **Refactoring**:
    - Comparing the cut and look to the customer's desired style
    - Making adjustments to meet the desired style
4. **Repeating**:
    - **Test first**: Determining the measurements for the suit
    - **Make the cuts**: Measuring the fabric and making the cuts
    - **Refactor**: Making changes based on the reviews

The preceding steps are an example of a TDD approach. The measurements must be taken before the tailor can start cutting the raw material. For a moment, imagine that the tailor didn't use a test-driven approach and didn't use a measuring tape (testing tool). It would be ridiculous if the tailor started cutting before measuring.

As a developer, do you “cut before measuring”? Would you trust a tailor without a measuring tape? How would you feel about a developer who doesn't test?

## Measure twice, cut once

The tailor always starts with measurements. What would happen if the tailor made cuts before measuring? What would happen if the fabric was cut too short? How much extra time would go into the tailoring? Thus, measure twice, cut once.

Software developers can choose from an endless amount of approaches to use before starting development. One common approach is to work off a specification. A documented approach may help in defining what needs to be built; however, without tangible criteria for how to meet a specification, the actual application that gets developed may be completely different from the specification. With a TDD approach, every stage of the process verifies that the result meets the specification. Think about how a tailor continues to use a measuring tape to verify the suit throughout the process.

TDD embodies a test-first methodology. TDD gives developers the ability to start with a clear goal and write code that will directly meet a specification, so you can develop like a professional and follow the practices that will help you write quality software.

# Practical TDD with JavaScript

Let's dive into practical TDD in the context of JavaScript. This walk through will take us through the process of adding the multiplication functionality to a calculator.

Just keep the TDD life cycle, as follows, in mind:

- Test first
- Make it run
- Make it better

## Point out the development to-do list

A development to-do list helps organize and focus on tasks individually. It also helps provide a platform to list down ideas during the development process, which could be a single feature later on.

Let's add the first feature in the development to-do list–the add multiplication functionality:

*3 * 3 = 9*

The preceding list describes what needs to be done. It also provides a clear example of how to verify the multiplication *3 * 3 = 9.*

# Setting up the test suite

To set up the test, let's create the initial calculator in a file called `calculator.js`. It is initialized as an object as follows:

```
var calculator = {};
```

The test will be run through a web browser as a simple HTML page. So for that, let's create an HTML page and import `calculator.js` to test it and save the page as `testRunner.html`.

To run the test, let's open the `testRunner.html` file in your web browser.

The `testRunner.html` file will look like this:

```
<!DOCTYPE html>
<html>
<head>
  <title>Test Runner</title>
</head>
<body>

<script src="calculator.js"></script>
</body>
</html>
```

The test suite is ready for the project and the development to-do list for the features is ready as well. The next step is to dive into the TDD life cycle based on the feature list one by one.

# Test first

Though it's easy to write a multiplication function and it will work as its a pretty simple feature, as a part of practicing TDD, it's time to follow the TDD life cycle. The first phase of the life cycle is to write a test based on the development to-do list.

Here are the steps for the first test:

1. Open `calculator.js`.
2. Create a new function `multipleTest1` to test multiplying *3 * 3*, after that `calculator.js` file will look as follows:

```
function multipleTest1() {
    // Test
    var result = calculator.multiply(3, 3);
```

```
        // Assert Result is expected
        if (result === 9) {
            console.log('Test Passed');
        } else {
            console.log('Test Failed');
        }
    };

    multipleTest1();
```

The test calls a `multiply` function, which still needs to be defined. It then asserts that the results are as expected by displaying a pass or fail message.

Keep in mind that in TDD, you are looking at the use of the method and explicitly writing how it should be used. This allows you to define the interface based on a use case, as opposed to only looking at the limited scope of the function being developed.

The next step in the TDD life cycle is focused on making the test run.

# Make the test run

In this step, we will run the test, just as the tailor did with the suite. The measurements were taken in the test step, and now the application can be molded to fit the measurements.

The following are the steps to run the test:

1. Open `testRunner.html` on a web browser.
2. Open the JavaScript developer **Console** window in the browser.

The test will throw an error, which will be visible in the browser's developer console, as shown in the following screenshot:


The error thrown is expected as the calculator application calls a function that hasn't been created yet–`calculator.multiply`.

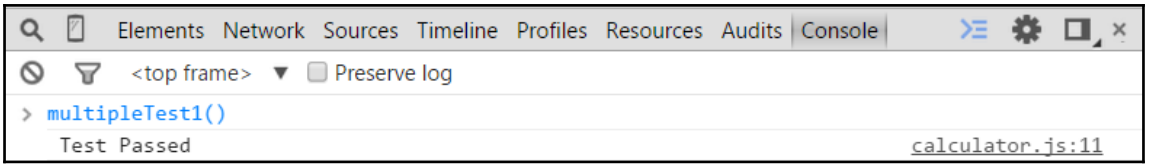
In TDD, the focus is on adding the easiest change to get a test to pass. There is no need to actually implement the multiplication logic. This may seem unintuitive. The point is that once a passing test exists, it should always pass. When a method contains fairly complex logic, it is easier to run a passing test against it to ensure that it meets the expectations.

What is the easiest change that can be made to make the test pass? By returning the expected value of `9`, the test should pass. Although this won't add the multiply feature, it will confirm the application wiring. In addition, after we have passed the test, making future changes will be easy as we have to simply keep the test passing!

Now, add the `multiply` function and have it return the required value of `9`, as illustrated:

```
var calculator = {
    multiply : function() {
        return 9;
    }
};
```

Now, let's refresh the page to rerun the test and look at the JavaScript console. The result should be as shown in the following screenshot:




Yes! No more errors. There's a message showing that test has been passed.

Now that there is a passing test, the next step will be to remove the hard coded value from the `multiply` function.

# Make the project better

The refactoring step needs to remove the hard coded `return` value from the `multiply` function, which we added as the easiest solution to pass the test, and add the required logic to get the expected result.

The required logic is as follows:

```
var calculator = {
    multiply : function(amount1, amount2) {
        return amount1 * amount2;
```

```
    }
};
```

Now, let's refresh the browser to rerun the tests; It will pass the test as it did earlier. Excellent! Now the `multiply` function is complete.

The full code of the `calculator.js` file for the `calculator` object with its test will look as follows:

```
var calculator = {
    multiply : function(amount1, amount2) {
        return amount1 * amount2;
    }
};

function multipleTest1() {
    // Test
    var result = calculator.multiply(3, 3);
    // Assert Result is expected
    if (result === 9) {
        console.log('Test Passed');
    } else {
        console.log('Test Failed');
    }
}

multipleTest1();
```

# Mechanism of testing

To be a proper TDD-following developer, it is important to understand some fundamental mechanisms of testing techniques and approaches to testing. In this section, we will walk through a couple of examples of testing techniques and mechanisms that will be leveraged in this book.

This will mostly include the following points:

- Testing doubles with **Jasmine** spies
- Refactoring the existing tests
- Building patterns

Here are the additional terms that will be used:

- **Function under test**: This is the function that is being tested. It is also referred to as system under test, object under test, and so on.
- **The 3 As (Arrange, Act, and Assert)**: This is a technique used to set up tests, first described by Bill Wake (`http://xp123.com/articles/3a-arrange-act-assert/`). The 3 As will be discussed further in `Chapter 2`, *Details of JavaScript Testing*.

# Testing with a framework

We have already seen a quick and simple way to perform tests on the calculator application, where we have set the test for the `multiply` method. But in real life, it will be more complex and a way larger application, where the earlier technique will be too complex to manage and perform. In that case, it will be handy and easier to use a testing framework. A testing framework provides methods and structures to test. This includes a standard structure to create and run tests, the ability to create assertions/expectations, the ability to use test doubles, and a lot more. This book uses Jasmine as the test framework. Jasmine is a behavior-driven testing framework. It is highly compatible with testing Angular applications. In `Chapter 2`, *Details of JavaScript Testing*, we will take an in-depth look at Jasmine.

The following example code is not exactly how it runs with the Jasmine test/spec runner, it's just about the idea of how the doubles work or how these doubles return the expected result. In `Chapter 2`, *Details of JavaScript Testing*, we will demonstrate exactly how this double should be used with the Jasmine spec runner.

# Testing doubles with Jasmine spies

A test double is an object that acts as and is used in place of another object. Jasmine has a test double function which is known as `spies`. A Jasmine spy is used with the `spyOn()` method.

Let's take a look at the following `testableObject` object that needs to be tested. Using a test double, we can determine the number of times `testableFunction` gets called.

The following is an example of a test double:

```
var testableObject = {
    testableFunction : function() { }
};
jasmine.spyOn(testableObject, 'testableFunction');
```

```
testableObject.testableFunction();
testableObject.testableFunction();
testableObject.testableFunction();

console.log(testableObject.testableFunction.count);
```

The preceding code creates a test double using a Jasmine spy (`jasmine.spyOn`). The following are some of the features that a Jasmine test double offers:

- The count of calls on a function
- The ability to specify a return value (stub a return value)
- The ability to pass a call to the underlying function (pass through)

Throughout this book, we will gain further experience in the use of test doubles.

## Stubbing a return value

The great thing about using a test double is that the underlying code of a method does not have to be called. With a test double, we can specify exactly what a method should return for a given test.

Consider the following example of an object and a function, where the function returns a string:

```
var testableObject = {
    testableFunction : function() { return 'stub me'; }
};
```

The preceding object, `testableObject`, has a function, `testableFunction`, that needs to be stubbed.

So, to stub the single return value, it will need to chain the `and.returnValue` method and will pass the expected value as `param`.

Here is how to spy chain the single return value to stub it:

```
jasmine.spyOn(testableObject, 'testableFunction')
.and
.returnValue('stubbed value');
```

Now, when `testableObject.testableFunction` is called, `stubbed value` will be returned.

Consider the following example of the preceding single `stubbed value`:

```
var testableObject = {
    testableFunction : function() { return 'stub me'; }
};
//before the return value is stubbed
Console.log(testableObject.testableFunction());
//displays 'stub me'

jasmine.spyOn(testableObject,'testableFunction')
.and
.returnValue('stubbed value');

//After the return value is stubbed
Console.log(testableObject.testableFunction());
//displays 'stubbed value'
```

Similarly, we can pass multiple returned values as in the preceding example.

Here is how to spy chain the multiple return values to stub them one by one:

```
jasmine.spyOn(testableObject, 'testableFunction')
.and
.returnValues('first stubbed value', 'second stubbed value', 'third stubbed
value');
```

So, for every call of `testableObject.testableFunction`, it will return the stubbed value in order until it reaches the end of the return value list.

Consider the example of the preceding multiple stubbed values:

```
jasmine.spyOn(testableObject, 'testableFunction')
.and
.returnValue('first stubbed value', 'second stubbed value', 'third stubbed
value');

//After the is stubbed return values
Console.log(testableObject.testableFunction());
//displays 'first stubbed value'
Console.log(testableObject.testableFunction());
//displays 'second stubbed value'
Console.log(testableObject.testableFunction());
//displays 'third stubbed value'
```

# Testing arguments

A test double provides insights into how a method is used in an application. As an example, a test might want to assert what arguments a method was called with or the number of times a method was called. Here is an example function:

```
var testableObject = {
    testableFunction : function(arg1, arg2) {}
};
```

The following are the steps to test the arguments with which the preceding function is called:

1. Create a spy so that the arguments called can be captured:

```
jasmine.spyOn(testableObject, 'testableFunction');
```

2. Then, to access the arguments, run the following:

```
//Get the arguments for the first call of the function
var callArgs = testableObject.testableFunction
.call.argsFor(0);

console.log(callArgs);
//displays ['param1', 'param2']
```

Here is how the arguments can be displayed using `console.log`:

```
var testableObject = {
    testableFunction : function(arg1, arg2) {}
};
//create the spy
jasmine.spyOn(testableObject, 'testableFunction');

//Call the method with specific arguments
  testableObject.testableFunction('param1', 'param2');

//Get the arguments for the first call of the function
var callArgs = testableObject.testableFunction.call.argsFor(0);

console.log(callArgs);
//displays ['param1', 'param2']
```

# Refactoring

Refactoring is the act of restructuring, rewriting, renaming, and removing code in order to improve the design, readability, maintainability, and overall aesthetics of a piece of code. The TDD life cycle step of *making the project better* is primarily concerned with refactoring. This section will walk us through a refactoring example.

Take a look at the following example of a function that needs to be refactored:

```
var abc = function(z) {
    var x = false;
    if(z > 10)
        return true;
    return x;
}
```

This function works fine and does not contain any syntactical or logical issues. The problem is that the function is difficult to read and understand. Refactoring this function will improve its naming, structure, and definition. The exercise will remove the masquerading complexity and reveal the function's true meaning and intention.

Here are the steps:

1. Rename the function and variable names to be more meaningful, that is, rename `x` and `z` so that they make sense:

```
var isTenOrGreater = function(value) {
    var falseValue = false;
    if(value > 10)
        return true;
    return falseValue;
}
```

   Now, the function can easily be read and the naming makes sense.

2. Remove any unnecessary complexity. In this case, the `if` conditional statement can be removed completely, as follows:

```
var isTenOrGreater = function(value) {
    return value > 10;
};
```

3. Reflect on the result.

At this point, the refactoring is complete, and the function's purpose should jump out at you. The next question that should be asked is: “Why does this method exist in the first place?”.

This example only provided a brief walk-through of the steps that can be taken to identify the issues in code and how to improve them. Other examples will be given throughout this book.

# Building with a builder

These days, the design pattern is kind of a common practice, and we follow the design patterns to make life easier. For the same reason, the builder pattern will be followed here.

The builder pattern uses a `builder` object to create another object. Imagine an object with 10 properties. How will test data be created for every property? Will the object have to be recreated in every test?

A `builder` object defines an object to be reused across multiple tests. The following code snippet provides an example of the use of this pattern. This example will use the `builder` object in the `validate` method:

```
var book = {
    id : null,
    author : null,
    dateTime : null
};
```

The `book` object has three properties: `id`, `author`, and `dateTime`. From a testing perspective, we would want the ability to create a valid object, that is, one that has all the fields defined. We may also want to create an invalid object with missing properties, or we may want to set certain values in the object to test the validation logic. Like here `dateTime` is an actual date time, which should assign by builder object.

Here are the steps to create a builder for the `bookBuilder` object:

1. Create a builder function, as shown here:

```
var bookBuilder = function() {};
```

2. Create a valid object within the builder, as follows:

```
var bookBuilder = function() {
    var _resultBook = {
```

```
        id: 1,
        author: 'Any Author',
        dateTime: new Date()
    };
}
```

3. Create a function to return the built object:

```
var bookBuilder = function() {
    var _resultBook = {
        id: 1,
        author: "Any Author",
        dateTime: new Date()
    };
    this.build = function() {
        return _resultBook;
    }
}
```

4. As illustrated, create another function to set the `_resultBook` author field:

```
var bookBuilder = function() {
    var _resultBook = {
        id: 1,
        author: 'Any Author',
        dateTime: new Date()
    };
    this.build = function() {
        return _resultBook;
    };
    this.setAuthor = function(author){
        _resultBook.author = author;
    };
};
```

5. Change the function definition so that calls can be chained:

```
this.setAuthor = function(author) {
    _resultBook.author = author;
    return this;
};
```

6. A setter function will also be created for `dateTime`, as shown here:

```
this.setDateTime = function(dateTime) {
    _resultBook.dateTime = dateTime;
    return this;
};
```

Now, `bookBuilder` can be used to create a new book, as follows:

```
var bookBuilder = new bookBuilder();

var builtBook = bookBuilder.setAuthor('Ziaul Haq')
.setDateTime(new Date())
.build();
console.log(builtBook.author); // Ziaul Haq
```

The preceding builder can now be used throughout our tests to create a single consistent object.

Here is the complete builder for reference:

```
var bookBuilder = function() {
    var _resultBook = {
        id: 1,
        author: 'Any Author',
        dateTime: new Date()
    };
    this.build = function() {
        return _resultBook;
    };

    this.setAuthor = function(author) {
        _resultBook.author = author;
        return this;
    };
    this.setDateTime = function(dateTime) {
        _resultBook.dateTime = dateTime;
        return this;
    };
};
```

Let's create the `validate` method to validate the created book object from the builder:

```
var validate = function(builtBookToValidate){
    if(!builtBookToValidate.author) {
        return false;
    }
    if(!builtBookToValidate.dateTime) {
        return false;
    }
    return true;
};
```

Let's start by creating a valid book object with the builder by passing all the required information, and if this is passed via the `validate` object, this should show a valid message:

```
var validBuilder = new bookBuilder().setAuthor('Ziaul Haq')
.setDateTime(new Date())
.build();

// Validate the object with validate() method
if (validate(validBuilder)) {
    console.log('Valid Book created');
}
```

In the same way, let's create an invalid book object via the builder by passing some null value in the required information. And by passing the object to the `validate` method, it should show the message explaining why it's invalid:

```
var invalidBuilder = new bookBuilder().setAuthor(null).build();

if (!validate(invalidBuilder)) {
    console.log('Invalid Book created as author is null');
}

var invalidBuilder = new bookBuilder().setDateTime(null).build();

if (!validate(invalidBuilder)) {
    console.log('Invalid Book created as dateTime is null');
}
```


**Downloading the example code**
You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books that you have purchased. If you have purchased this book from elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

# Self-test questions

Q1. A test double is another name for a duplicate test.

1. True
2. False

Q2. TDD stands for test-driven development.

1. True
2. False

Q3. The purpose of refactoring is to improve code quality.

1. True
2. False

Q4. A test object builder consolidates the creation of objects for testing.

1. True
2. False

Q5. The 3 As are a sports team.

1. True
2. False

# Summary

This chapter provided an introduction to TDD. It discussed the TDD life cycle (test first, make it run, and make it better) and these steps can be used by anybody for TDD approach similar to how we have saw being used by a tailor. Finally, it looked over some of the testing techniques that will be discussed throughout this book, including test doubles, refactoring, and building patterns.

Although TDD is a huge topic, this book is solely focused on the TDD principles and practices to be used with Angular.

In the next chapter, we will know details about JavaScript testing.

# 2
# Details of JavaScript Testing

Practicing TDD is a great way to get good quality software with satisfactory accuracy, even with fewer people. For web applications, JavaScript has become the most popular scripting language and it has become a challenge to test JavaScript code. Browser-based testing is actually a time killer and is difficult to follow for TDD, but then the solution to this comes with some cool tools that support automated testing for JavaScript. Most web application projects were limited to unit tests only, and without automated test tools, end-to-end tests or functional tests were almost impossible.

A lot of tools and frameworks focusing on JavaScript testing are coming out, which serve different solutions, making developers' lives easy. Besides inventing new JavaScript frameworks, the developer's community invented some tool sets to make testing easy. Like the Angular team, they come with cool tools like **Karma**. We also have the duplication of testing frameworks or tools, where both solve similar problems but in different ways. Which tools or frameworks to choose is up to the developer; they have to choose the tool that suits their requirements best.

In this chapter, we will cover the following:

- A brief about automated testing
- Different types of testing focused on JavaScript
- A brief idea about some testing tools and frameworks

# The craft of JavaScript testing

We all know that JavaScript is a dynamically typed, interpreted language. Therefore, there are no compilation steps that help you figure out errors, unlike other compiled languages similar to Java. So, a JavaScript developer should allocate more time to test code. However, life is easier now, as a developer can cover testing with minimal steps and time using the latest tool techniques. It's a part of an automated test, where the code will be automatically tested whenever it changes. In that process, a test could be a task running in the background, which could be integrated into the IDE or the CLI, and it will provide the test result during development.

In the subsequent sections, we will discuss how to automate the test process in multiple browsers with a test runner and a headless browser.

# Automated testing

Testing is fun, and writing a test will make the code better; it's a good practice, but the procedural manual testing is a bit time consuming, error prone, and irreproducible. In this process, there is a need to write the test spec, change the code to pass the test, refresh the browser to get the result, and repeat this process several times. As a programmer, it's kind of boring to repeat the same things.

Apart from being monotonous, it also slows down the development process a lot, which demotivates developers from practicing TDD. So, when the manual process slows down progress, we have to look for some automated process to do the job and save time for other tasks that could add more business value.

So, it would be great to have some tools or techniques that can help programmers get rid of these repetitive and boring manual steps that slow down the process and get things done automatically, faster, and save time to make them more valuable for the business. Fortunately, there are some tools to automate these tests. We will cover more about those tools and techniques, but not in this section.

Besides the issue of slowing down the development process, another important point comes in view when we talk about the testing features, and that is the cross-browser compatibility issue. As web applications should run perfectly on modern platforms and browsers, and it's almost impossible to test them one-by-one manually, automated testing could be the solution with a web driver and headless browser.

Let's recap the basic test flow that we explained in the previous chapter–test it, make it run, and make it better. To make this process automatic, a developer can implement the tool set in the CLI or even in the development IDE, and these tests will run continuously in a separate process without any input from the developer.

Let's think of a registering or signing up feature for any application, where we have to manually fill up the form and click on the submit button every time we want to test the feature and repeat the process by changing the data. This is actually known as a functional test (which we will discuss at the end of this chapter). To perform these processes automatically, we will use the tool set (the test runner, web driver, and headless browser) in the CLI and complete the process with a single command with some parameters.

Testing JavaScript in automated testing is not a new concept, indeed, it is the most commonly used automated browser. Selenium was invented for this in 2004, and after that, a lot of tools have emerged, including PhantomJS, Karma, Protractor, and CasperJS. In this chapter, we will discuss some of them.

# Types of testing

In TDD, developers have to follow a flow to fulfill the goal of the test. On this flow, every step has an individual goal of testing. For example, some tests are written just to test the behavior of each function in several ways and some are for testing the flow of a module/feature. Based on that, we will discuss two major types of testing here. They are as follows:

- **Unit testing**: This is mostly used for behavior tests.
- **End-to-end testing**: This is mostly known as e2e testing and is used for functional tests.

# Unit testing

**Unit testing** is a software development process, where the smallest testable part of any application is individually called a unit, and the behavior of that small part should be testable in isolation, without any dependency on the other parts. If we think of a JavaScript application as software, then every individual method/function of that application which has a specific behavior will be a unit of code. The behavior of these methods or units of code should be testable in an isolated way.

An important point about unit testing is that any unit of code should run/be testable in isolation and should run in any order, which means that if unit testing runs successfully in any application, it represents the isolation of the components or modules of that application.

For example, we had already shown a small test example in the previous chapter about how to get a method tested; though we showed that without using any test framework, the idea is the same. We called the method by passing some parameters, got a result of that method, and then we compared the result with the expected value.

Typically, we will write such tests using a unit testing framework of our choice. There are many testing frameworks and tools now, and we have to decide and pick the best one based on our requirements. The most commonly used frameworks are Jasmine, Mocha, and QUnit. We will discuss these tools in depth in this chapter and real-life examples will be covered in the subsequent chapters.

Tests should run fast and be automated with a clear output. For example, you can verify that if a function is called with particular arguments, it should return an expected result.

Unit testing can run the test anytime, such as in the following instances:

- From the very beginning of the development process, even with a failing test
- After completing the development of any feature to verify that the behavior is correct
- After modifying any existing feature to verify that the behavior hasn't changed
- After adding a new feature in the existing application, we need to verify that the new feature is isolated and it's not breaking any other feature

# End-to-end testing

End-to-end testing is a methodology used to test whether the flow of an application is performing as designed from start to finish. For example, if a user clicks on a product from a product list, it should prompt the modal to display detailed information of the selected product. In this case, the product/project owner will define the project requirements step by step in the specification. The project will be tested based on the specification's workflow after the development process. This is called the function/flow test and is another name for an end-to-end test.

Besides unit testing, end-to-end testing is important for confirming that individual components are working together as an application, passing information, and communicating among them. The main difference with unit testing is that it does not test any components in isolation; instead, it is a combined test of the flow with all the dependent components together.

Consider a registration module where users should provide some valid information to complete the registration, and the function/flow test for that module/application should follow some steps to complete the testing.

The steps are as follows:

1. Load/compile the form
2. Get the DOM of the form's elements
3. Trigger the click event of the submit button
4. Collect the value from the input fields for validation
5. Validate the input fields
6. Call the fake API to store data

In every step, there will be some result which will be compared to the expected result set.

These kinds of functional/flow tests can be tested manually by a person filling out the forms by clicking on the buttons for the next steps, completing the application flow, and comparing the result with the specification that is defined earlier in the implementation process.

However, there are some techniques available to do this functional/flow testing in an automated way without getting input from any person, which is known as end-to-end testing. To make this test process easier, there are a few tools available; the ones used most commonly are Selenium, PhantomJS, and Protractor. These tools can easily integrate with any application test system. In this chapter, we will discuss these test tools in a bit more detail, and in the subsequent chapters, we will integrate these in an application's test suite.

# Testing tools and frameworks

Knowing what the different testing tools are is half the battle. A few of them are very important to know in depth for Angular testing; we will learn them in detail throughout this book. However, in this section, we will learn about some well-known tools and frameworks that are used in different web applications for various kinds of testing and approaches. They are as follows:

- **Karma**: This is the test runner for JavaScript
- **Protractor**: This is the end-to-end testing framework
- **Jasmine**: This refers to the behavior-driven JavaScript testing framework
- **Mocha**: This is the JavaScript testing framework
- **QUnit**: This stands for the unit testing framework
- **Selenium**: This is the tool that automates the web browsers
- **PhantomJS**: This is the headless webkit browser

# Karma

Before discussing what Karma is, it is best to discuss what it isn't. It isn't a framework to write tests; it is a test runner. What this means is that Karma gives us the ability to run tests in several different browsers in an automated way. In the past, developers had to perform manual steps to do this, including the following:

- Opening up a browser
- Pointing the browser to the project URL
- Running the tests
- Confirming that all the tests have passed
- Making changes
- Refreshing the page

With Karma, automation gives a developer the ability to run a single command and determine whether an entire test suite has passed or failed. From a TDD perspective, this gives us the ability to find and fix failing tests quickly.

Some of the pros of using Karma as compared to a manual process are as follows:

- Ability to automate tests in multiple browsers and devices
- Ability to watch files
- Online documentation and support

- Does one thing–running JavaScript tests–and does it well
- Makes it easy to integrate with a continuous integration server

Disadvantage of using Karma:

- Requires an additional tool to learn, configure, and maintain

Automating the process of testing and using Karma is extremely advantageous. In the TDD journey through this book, Karma will be one of our primary tools. We will learn about Karma in detail in `Chapter 3`, *The Karma Way*.

# Protractor

Protractor is an end-to-end testing tool that allows developers to mimic user interactions. It automates the testing of functionality and features through the interaction of a web browser. Protractor has specific methods to assist in the testing of Angular, but they are not exclusive to Angular.

Some of the pros of using Protractor are as follows:

- Configurable to test multiple environments
- Easy integration with Angular
- Syntax and testing can be similar to the testing framework chosen for unit testing

Disadvantage of using Protractor:

- Its documentation and examples are limited

For end-to-end testing of the examples in this book, Protractor will be our main framework. Protractor will be further introduced in detail in `Chapter 4`, *End-to-End Testing with Protractor*.

# Jasmine

Jasmine is a behavior-driven development framework for testing JavaScript code. It can be easily integrated and run for websites and is agnostic to Angular. It provides spies and other features. It can also be run on its own without Karma. In this chapter, we will learn details of the commonly used built-in global functions of Jasmine and will see how the Jasmine test suite serves the testing requirements for a web application. Also, throughout this book, we will use Jasmine as our testing framework.

Some of the pros of using Jasmine are as follows:

- Default integration with Karma
- Provides additional functions to assist with testing, such as test spies, fakes, and the pass-through functionality
- Cleans readable syntax that allows tests to be formatted in a way that relates to the behavior being tested
- Integration with several output reporters

The following are some cons of using Jasmine:

- No file-watching feature is available when running tests. This means that tests have to be rerun by the user as they change.
- The learning curve can be steep for all the Protractor methods and features.

# Mocha

Mocha is a testing framework originally written for Node.js applications, but it supports browser testing as well. It is very similar to Jasmine and mirrors much of its syntax. The main difference with Mocha is that it can't run standalone as a test framework–it needs some plugin and library to run as a test framework, while Jasmine is standalone. It's more configurable and flexible to use.

Let's discuss some of the pros of Mocha:

- Easy to install
- Good documentation available
- Has several reporters
- Plugs in with several node projects

Here are a few cons:

- Separate plugins/modules are required for assertions, spies, and so on
- Additional configuration required to use it with Karma

# QUnit

QUnit is a powerful, easy-to-use JavaScript unit test suite. It's used by jQuery and the jQuery UI and jQuery Mobile projects and is capable of testing any generic JavaScript code. QUnit focuses on testing JavaScript in the browser, while providing as much convenience to the developer as possible.

Some of the pros of QUnit:

- Easy to install
- Good documentation available

Here is a con of using QUnit:

- Mostly developed for jQuery, not good for use with other frameworks

# Selenium

Selenium (`http://www.seleniumhq.org/`) defines itself as follows:

> *"Selenium automates browsers. That's it!"*

Automation of browsers means that developers can interact with browsers easily. They can click on buttons or links, enter data, and so on. Selenium is a powerful toolset that, when used and set up properly, has lots of benefits; however, it can be confusing and cumbersome to set it up.

Some of the pros of Selenium are as follows:

- Large feature set
- Distributed testing
- SaaS support through services such as **Sauce Labs** (`https://saucelabs.com/`)
- Documentation and resources available

Here are some cons of Selenium:

- Has to be run as a separate process
- Several steps to configure

As Protractor is a wrapper around Selenium, it won't be discussed in detail.

# PhantomJS

PhantomJS is a headless WebKit scriptable with a JavaScript API. It has *fast* and *native* support for various web standards; DOM handling, CSS selector, JSON, Canvas, and SVG. PhantomJS is used in the test workflow.

Simply put, PhantomJS is a browser that runs headlessly (that is, doesn't draw out the screen). The benefits that it brings are speed–if you're controlling an actual program on your computer, you've a certain overhead in booting up the browser, configuring a profile, and so on.

PhantomJS is not meant to replace a testing framework; it will work in conjunction with one.

# The choice is ours

As we have seen, there are a lot of toolsets and frameworks for testing JavaScript projects:

- For the assertion framework, we will go with Jasmine as Angular itself uses Jasmine as assertions; but in some cases, mostly for Node.js projects, Mocha is fun as well
- As long as we focus on the automated test suite, the test runner is of most importance to us, and nothing can be compared to Karma when it's about an Angular project
- For end-to-end testing, Protractor is the best framework, and we will use that in this chapter.
- As long as it's end-to-end testing, it must be automated, and Selenium is here to automate the browser for us.
- It's important to run tests as cross-browser support, and PhantomJS is here for us to serve as a headless browser.

# Say hello to Jasmine test suite

As long as we have to use a testing framework to build a test suite, there are some basic and common assertions on all frameworks. It's important to understand those assertions and spies and when to use them.

In this section we will explain the assertions and spies from Jasmine, as Jasmine will be our testing framework throughout the book.

# Suites

Any test suite begins with a global Jasmine `describe` function, which receives two parameters. The first one is a string and the second one is a function. The string is the suite name/title, and the function is for the code block that will be implemented in the suite.

Consider the following example:

```
describe("A sample test suite to test jasmine assertion", function() {
   // .. implemented code block
});
```

# Spec

Any spec defined with Jasmine's global `it` function, similar to the suite's that receives two params, involves the first one being a string and the second one being a function. The string is the spec name/title and the function is for the code block that will be implemented in the spec. Take a look at the following example:

```
describe("A sample test suite to test jasmine assertion", function() {
    var a;
    it("Title for a spec", function() {
        // .. implemented code block
    });
});
```

# Expectation

Any expectation defined with an `expect` function, which receives one param value that is called actual. This function is a chain with the matcher function, which takes an expected value as a parameter to match with the actual value.

There are a few commonly used matchers; all of them implement a Boolean comparison between the actual value and the expected value. Any matcher can evaluate a negative value by chaining the `expect` method with a `not` keyword.

Some common matchers are `toBe`, `toEqual`, `toMatch`, `toBeNull`, `toBeDefined`, `toBeUndefined`, and `toContain`.

Consider the given example:

```
describe("A sample test suite to test jasmine assertion", function() {
    var a, b;
    it("Title for a spec", function() {
        var a = true;
        expect(a).toBe(true);
        expect(b).not.toBe(true);
    });
});
```

# Setup and teardown

To improve the test suite by DRY (Don't repeat yourself) up duplicated setup and teardown code, Jasmine provides some global functions for the setup and teardown. Those global functions (`beforeEach`, `afterEach`, and so on) are as follows and they run as the name implies.

Every function runs against a test spec. Jasmine's global setup and the teardown functions are `beforeEach`, `afterEach`, `beforeAll`, and `afterAll`.

Consider the following example:

```
describe("A sample test suite to test jasmine assertion", function() {
    var a=0;
    beforeEach(function() {
        a +=1;
    });
    afterEach(function() {
        a =0;
    });
    it("Title for a spec 1", function() {
        expect(a).toEqual(1);
    });
    it("Title for a spec 2", function() {
        expect(a).toEqual(1);
        expect(a).not.toEqual(0);
    });
});
```

# Spies

Spies are test double functions in Jasmine; they can stub any function and track calls on it and to all its arguments. There are a few matchers around to track if any spy has been called or not. These are `toHaveBeenCalled`, `toHaveBeenCalledTimes`, and so on.

There are some useful chained methods used with spy, such as `returnValue`/`returnValues`, which will return one or more values when called by chaining with spy. There are a few more similar useful methods, such as `callThrough`, `call`, `stub`, `call.allArgs`, `call.first`, and `call.reset`.

Consider the following example:

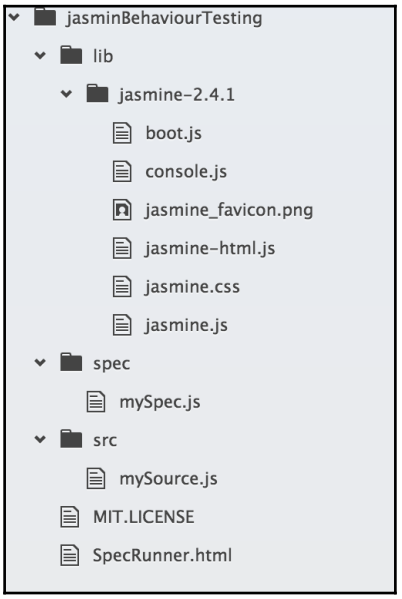
```
describe("A sample test suite to test jasmine assertion", function() {
    var myObj, a, fetchA;
    beforeEach(function() {
        myObj = {
            setA: function(value) {
                a = value;
            },
            getA: function(value) {
                return a;
            },
        };
        spyOn(myObj, "getA").and.returnValue(789);
        myObj.setA(123);
        fetchA = myObj.getA();
    });

    it("tracks that the spy was called", function() {
         expect(myObj.getA).toHaveBeenCalled();
    });
    it("should not affect other functions", function() {
        expect(a).toEqual(123);
    });
    it("when called returns the requested value", function() {
        expect(fetchA).toEqual(789);
    });
});
```

# The test suite of Jasmine

In the preceding section, we looked over some commonly used assertions that all test frameworks, including Jasmine, use in any kind of test suite.

Though in this book, we will build an automated test suite for Angular testing, let's try some assertion in a Jasmine test suite and see how it works. This sample test suite will give us some hands on experience of how the assertions work on a test suite.

For this test suite, we will use Jasmine's sample spec runner project (which is available in the Jasmine downloaded bundle in the example), and the project's folder structure will look as follows:

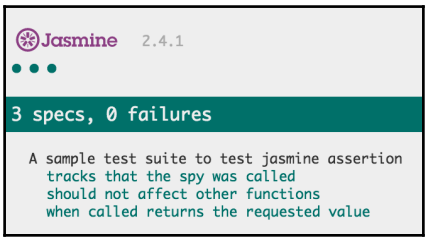



Let's take a quick look at the files that we have to update in the Jasmine's sample `SpecRunner` project:

```
SpecRunner.html:
<!DOCTYPE HTML>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Jasmine Spec Runner v2.4.1</title>
    <link rel="shortcut icon" type="image/png"
href="lib/jasmine-2.4.1/jasmine_favicon.png">
    <link rel="stylesheet" type="text/css"
```

```
href="lib/jasmine-2.4.1/jasmine.css">

    <script type="text/javascript"
src="lib/jasmine-2.4.1/jasmine.js"></script>
    <script type="text/javascript" src="lib/jasmine-2.4.1/jasmine-
html.js"></script>
    <script type="text/javascript"
src="lib/jasmine-2.4.1/boot.js"></script>
    <!-- include source files here... -->
    <script type="text/javascript" src="src/mySource.js"></script>
    <!-- include spec files here... -->
    <script type="text/javascript" src="spec/mySpec.js"></script>
</head>
<body>
</body>
</html>

src/mySource.js:
var a,
myObj = {
    setA: function(value) {
        a = value;
    },
    getA: function(value) {
        return a;
    },
};

Spec/mySpec.js:
describe("A sample test suite to test jasmine assertion", function() {
    var fetchA;
    beforeEach(function() {
        spyOn(myObj, "getA").and.returnValue(789);
        myObj.setA(123);
        fetchA = myObj.getA();
    });

    it("tracks that the spy was called", function() {
         expect(myObj.getA).toHaveBeenCalled();
    });
    it("should not affect other functions", function() {
        expect(a).toEqual(123);
    });
    it("when called returns the requested value", function() {
        expect(fetchA).toEqual(789);
    });
});
```

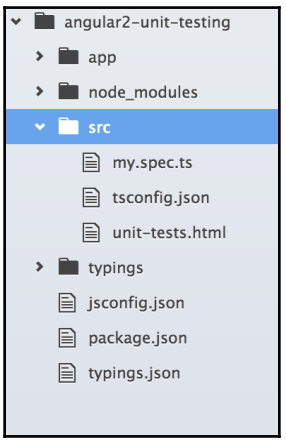
As long as it's a browser-based test suite, we have to point `SpecRunner.html` to a web browser to get the test result. We will have all the tests passed and our test result will look as shown in the following screenshot:




# The Jasmine test suite for Angular

In the preceding example, we saw a Jasmine test suite for JavaScript testing, but what about for Angular, how should that look? Actually, there is no direct answer as, for the Angular project test suite, we will not use a browser-based test suite; we have a test runner with Karma for the test suite. But as we are familiar with the browser-based Jasmine test suite in the preceding example, let's see what that will look like if we make a similar one for the Angular project.

We will have to add a subfolder as `src` in the Angular project for the test spec, and then the project's folder structure will look like this:



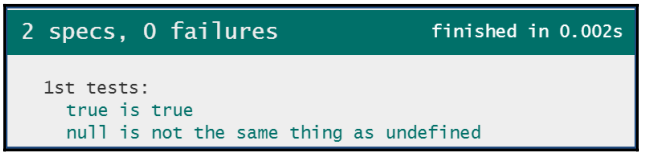
In the Angular project, we will use TypeScript rather than plain JavaScript as Angular officially suggests using TypeScript. So, we hope that we all know the TypeScript syntax and know how to compile to JS.
In this book, for the Angular test suite, we will use SystemJS as the module loader, as Angular officially suggests it; we will take a look at SystemJS.
This sample Angular test suite is just to show how easily we can make a test suite for an Angular project, though it's not following the best practices and the best way to load the modules.
In `Chapter 3`, *The Karma Way*, we will update this test suite with real-life examples and use SystemJS as the module loader.
There is a seed project from the Angular team in GitHub called as `Angular2-seed` to bootstrap any Angular project with test suite; we will follow that one for our real Angular project.

If we look at the folder structure, it's almost the same as the previous one, and there are minimum changes in the spec file as well; the only change in spec is with TypeScript:

```
src/unit-tests.html:

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <title>NG2 App Unit Tests</title>
    <link rel="stylesheet" href="node_modules/jasmine-core/lib/jasmine-
core/jasmine.css">
    <script src="node_modules/jasmine-core/lib/jasmine-
core/jasmine.js"></script>
    <script src="node_modules/jasmine-core/lib/jasmine-core/jasmine-
html.js"></script>
    <script src="node_modules/jasmine-core/lib/jasmine-
core/boot.js"></script>
    <script src="../app/mysource.js"></script>
    <script src="my.spec.js"></script>

</head>
<body>
</body>
</html>

app/mysource.ts:
export class Source {
    // ...
}

src/my.spec.ts:
```

```
describe('1st tests', () => {
    it('true is true', () => expect(true).toEqual(true));

    it('null is not the same thing as undefined',
        () => expect(null).not.toEqual(undefined)
    );
});
```

As this is also a browser-based test suite, we have to point `unit-tests.html` in a web browser to get the test result. We will have all the tests passed, and our test result will look like this:




# Self-test questions

Self-test questions will help you further test your knowledge of using TDD for JavaScript application testing.

Q1. End-to-end testing means what kind of test?

- Functional test
- Behavior test

Q2. Protractor is a Unit testing framework.

- True
- False

Q3. PhantomJS is a type of browser.

- True
- False

Q4. What is QUnit a test framework for?

- jQuery
- Angular
- NodeJS

Q5. Setup and teardown is a feature of Jasmine.

- True
- False

# Summary

In this chapter, we reviewed different mechanisms of testing TDD and covered automated testing. We reviewed different kinds of JavaScript testing frameworks and tools and reviewed the uses of those frameworks with their pros and cons. We also reviewed some common assertions from Jasmine and tried, hands on, how they worked.

In the next chapter, we will learn about Karma and see how that works with the Angular test suite.

# 3
# The Karma Way

JavaScript testing has hit the mainstream, thanks to Karma. Karma makes it seamless to test JavaScript. Angular was created around testing.

In this chapter, we will learn a few things about Karma, including the following:

- The origin of Karma
- Why and how Karma will work with an Angular project
- The Karma configuration and implementation with an Angular project
- An overview of **Travis CI**

## The birth of Karma

When picking up a new tool, it is important to understand where it comes from and why it is built. This section will give us a background of the origin of Karma.

## The Karma difference

Karma was created by Vojta Jína. The project was originally called **Testacular**. In Vojtech Jína's thesis, he discusses the design, purpose, and implementation of Karma.

In his thesis (*JavasScript Test Runner*, `https://github.com/karma-runner/karma/raw/master/thesis.pdf`), he describes Karma as follows:

> *"...a test runner that helps web application developers to be more productive and effective by making automated testing simpler and faster. In fact, I have a much higher ambition and this thesis is only a part of it – I want to promote Test Driven Development (TDD) as "the" way to develop web applications, because I believe it is the most effective way to develop high quality software."*

Karma has the ability to easily and automatically run JavaScript unit tests on real browsers. Traditionally, tests were run by launching a browser manually and checking for results by continually clicking on the refresh button. This method was awkward and often resulted in developers limiting the number of tests that were written.

With Karma, a developer can write a test in almost any standard test framework, choose a browser to run against, set the files to watch for changes, and bam! We have continuous automated testing. We have to simply check the output window for failed or passed tests.

# The importance of combining Karma and Angular

Karma was built for AngularJS. Before Karma, there was a lack of automated testing tools for web-based JavaScript developers.

Remember that Karma is a test runner, not a test framework. Its job is to run tests and report which tests will pass or fail. Why is this helpful? A test framework is where you will write your tests. Apart from doing this, you will need to be focused on running the tests easily and seeing the results. Karma easily runs tests across several different browsers. It also has some other features, such as file watching, which will be discussed in detail later in the book.

# Installing Karma

It's time to start using Karma. Installations and applications are constantly changing. The following guide is intended to be brief; you can go to the Karma website at `http://karma-runner.github.io/` and find the latest instructions there.

The main focus of this section will be the specific configuration used in this book and not an in-depth installation guide.

# Installation prerequisites

To install Karma, we will need to have Node.js on our computer. Node.js runs on Google's V8 engine and allows JavaScript to be run on several operating systems.

Developers can publish node applications and modules using **NPM** (**Node Package Manager**). NPM allows developers to quickly integrate applications and modules into their applications.

Karma runs and is installed through the `npm` package; therefore, we need Node.js before we can use or install Karma. To install Node.js, go to `http://nodejs.org/` and follow the installation instructions.

Once we have Node.js installed, lets type the following command in the Command Prompt to install Karma:

```
$ npm install karma -g
```

The preceding command uses `npm` to install Karma globally using `-g`. This means that we can use Karma on the Command Prompt by simply typing the following:

```
$ karma --version
```

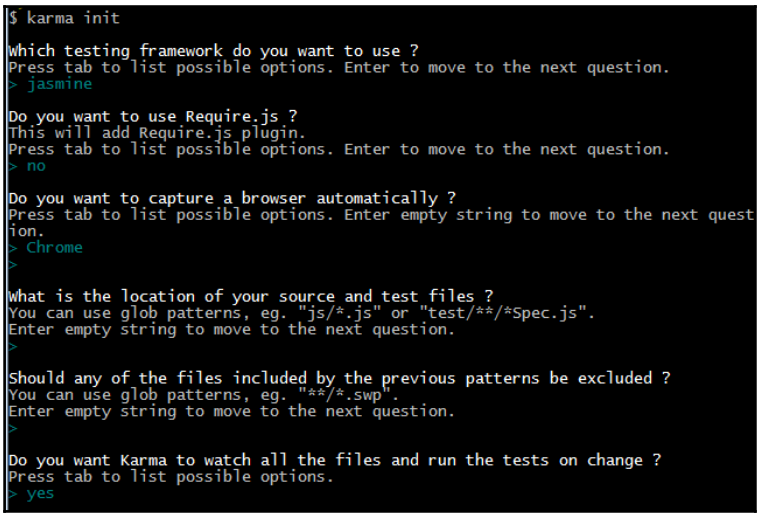
By default, installing Karma will install `karma-chrome-launcher` and `karma-jasmine` as dependencies. Ensure that these modules are installed globally as well.

# Configuring Karma

Karma comes equipped with an automated way to create a configuration file. To use the automated way, type the following command:

```
$ karma init
```

Here is a sample of the options chosen:




# Customizing Karma's configuration

The following instructions describe the specific configuration required to get Karma running for the project. Customization includes the test framework (Jasmine), the browser (Chrome) to test with, and the files to test. To customize the configuration, open `karma.conf.js` and perform the following steps:

1. Ensure that the enabled framework says `jasmine` using the following code:

```
frameworks: ['jasmine'],
```

2. Configure the `test` directory. Note that the following definition needs to include the tests that are required to be run along with any potential dependencies. The directory that will hold our tests is `/test/unit/`:

```
files: [
    'test/unit/**/*.js'
],
```

3. Set the test browser to Chrome, as follows. It will then be initialized and will run a popup after every test:

```
browsers: ['Chrome'],
```

# Confirming Karma's installation and configuration

To confirm Karma's installation and configuration, perform the following steps:

1. Run the following command to confirm that Karma starts with no errors:

```
$ karma start
```

2. The output should be something like this:

```
$ INFO [karma]: Karma v0.12.16 server started at
http://localhost:9876/
```

3. In addition to this, the output should state that no test files were found:

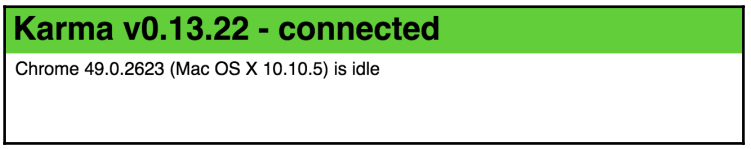
```
$ WARN [watcher]: Pattern "test/unit/**/*.js" does not
match any file.
```

   The output should do this along with a failed test message:

```
$ Chrome 35.0.1916 (Windows 7): Executed 0 of 0 ERROR
(0.016 secs / 0 secs)
```

An important point to note is that we will need to install `jasmine-core` globally on the system, or else Karma will not run successfully.

This is expected as no tests have been created yet . Continue to the next step if Karma starts, and we will see our Chrome browser with the following output:



# Common installation/configuration issues

If the Jasmine or Chrome launcher are missing, perform the following steps:

1. When running the test, an error might occur saying **missing Jasmine or Chrome Launcher**. If you get this error, type the following command to install the missing dependencies:

```
$ npm install karma-jasmine -g
$ npm install karma-chrome-launcher -g
```

2. Retry the test and confirm that the errors have been resolved.

   In some cases, you might not be able to install `npm_modules` globally using the `-g` command. This is generally due to permission issues on your computer. The following is what you need to do to provide permissions (sudo/administrator):

   1. The resolution is to install Karma directly in your project folder. Use the same command without `-g` to do this:

```
$ npm install karma
```

   2. Run Karma using the relative path:

```
$ ./node_modules/karma/bin/karma --version
```

Now that Karma is installed and running, it's time to put it to use.

# Testing with Karma

In this section, we will create a test to confirm that Karma is working as expected. To do this, perform the following steps:

1. Create the test directory. In the Karma configuration, tests were defined in the following directory:

```
files: [
    'test/unit/**/*.js'
],
```

2. Go ahead and create the `test/unit` directory.
3. Create a new `firstTest.js` file in the `test/unit` directory.

4. Write the first test as follows:

```
describe('when testing karma', function (){
  it('should report a successful test', function (){
    expect(true).toBeTruthy();
      });
});
```

The preceding test uses the Jasmine functions and has the following properties:

- `describe`: This provides a brief string description of the test suite, the things that will be tested.
- `it`: This provides a brief string of the specific assertion called a test spec
- `expect`: This provides a way to assert values
- `toBeTruthy`: This is one of the several properties of an expectation that can be used to make assertions

This test has no real value other than to confirm the output of a passing test.

Bam! Let's check our console window and see that Karma has executed our test. And our command line should say something like this:

```
$ INFO [watcher]: Added file "./test/unit/firstTest.js"
```

This output means that Karma automatically recognized that a new file was added. The next output should be something like this:

```
$ Chrome 35.0.1916 (Windows 7): Executed 1 of 1 SUCCESS (0.02 secs
/ 0.015 secs)
```

This means that our test has passed!

# Confirming the Karma installation

Now, the initial set up and configuration of Karma is complete. Here is a review of the steps:

1. We installed Karma through the `npm` command.
2. We initialized a default configuration through the `karma init` command.
3. Next, we configured Karma with Jasmine and a `test/unit` test directory.
4. We started Karma and confirmed that it could be opened with Chrome.
5. Then, we added a Jasmine test, `firstTest.js`, to our `test/unit` test directory.

6. Karma recognized that `firstTest.js` had been added to the test directory.
7. Finally, Karma executed our `firstTest.js` and reported our output.

With a couple of steps, we were able to see Karma running and executing tests automatically. From a TDD perspective, we can focus on moving tests from failing to passing without much effort. There is no need to refresh the browser; just check the command output window. Keep Karma running and all our tests and files will automatically be added and run.

In the subsequent sections, we will see how to apply Karma with a TDD approach. If you're okay with Karma so far and want to move on to Protractor, skip to the next chapter.

# Using Karma with Angular

Here, we will walk through a TDD approach to an Angular component. By the end of this chapter, we should be able to do the following:

- Feel confident about using Karma and its configuration
- Understand the basic components of a Jasmine test
- Start understanding the integration of a TDD approach in an Angular application

## Getting Angular

Angular installation is not possible via Bower; as it was with Angular1, it has to be installed via npm. Bootstrapping the Angular application is not as simple as Angular1 because Angular doesn't use plain JavaScript. It uses TypeScript or ES6 (ES2015), both of which need to be compiled to plain JavaScript before running them.

We believe that most of the developer are already aware of the Angular changes and how its compilation works. Just a quick recap–we will use TypeScript in our Angular example project here as long as Angular recommends that, though there is an option to use ES6 instead. We will use the node/npm tsc module to compile the TypeScript to plain JavaScript; node/npm will be our CLI tools as well to build/start a project and run the test.

A basic understanding of node/npm modules is required here, specifically, how the npm commands works.

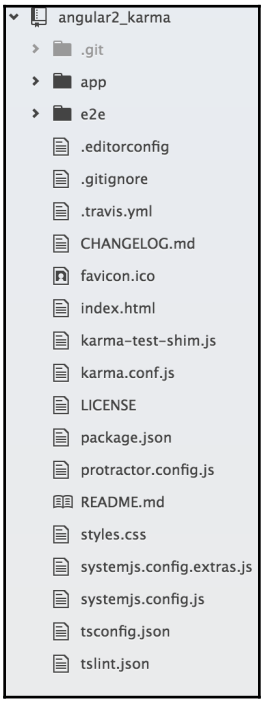
# Angular project

We will not demonstrate how to install Angular and how to build a project from scratch, as the Angular doc site shows that nicely. So, we will get a simple Angular project from the Angular teams' example and update that one for our implementation.

We will clone the `quickstart` project from the Angular GitHub repo and will start on that one. Hope we all have `git` installed globally besides the node/npm.

```
$ git clone https://github.com/angular/quickstart.git angular-
karma
```

This will copy the project locally as `angular-karma` and the folder structure will look as illustrated:




Let's proceed with it and get ready to run:

```
$ cd angular-karma
$ npm install
```

Here are a couple of steps to get ready with the example project. The `npm install` command will install the required modules for the project dependencies that are defined in the `package.json` file in the project root.

Then, we will run the project with `npm start`; this script, defined in `package.json`, is used to run the project in the local server.

Let's compile and run the project:

```
$ npm start
```

If all the required dependencies are installed, this command will compile the TypeScript to plain JavaScript and will run the project in the local server.

The project will launch in a browser and will look as follows:


If this sample project runs successfully, then we are good to go in the next step, where we will add a test spec, which will include Karma, and run those tests with the Karma runner.

## Getting ready

As we cloned the sample `quickstart` project, it already integrated and configured Karma in the project. For the purpose of learning, we would like to integrate Karma in the existing project.

To do so, we will have to remove the existing `karma.conf.js` file from the project root directory. Also, we will remove the Karma, Jasmine, and related modules from `node_modules`.

Interestingly, instead of doing it manually, we can easily create the basic Karma config file with a simple command. And with that command, it will ask some basic questions as we saw in the previous part of this chapter.

# Setting up Karma

To set up Karma in an Angular project, the first job is to create a `karma.conf.js` file in the project root directory. This file basically contains some configuration in key-value pairs.

Interestingly, instead of doing it manually, we can easily create the basic Karma config file with a simple command. And with that command, it will ask some basic questions as we saw in the previous part of this chapter:

```
$ karma init
```

Use the default answers. After `karma.conf.js` has been created in the current directory, open the configuration. A few configurations may need to change, mostly the one that is the definition of the files for Karma to use.

Use the following definition in the `files` section, which defines the files required to run the test:

```
files: [
        // System.js for module loading
      'node_modules/systemjs/dist/system.src.js',

      // Polyfills
      'node_modules/core-js/client/shim.js',
      'node_modules/reflect-metadata/Reflect.js',

      // zone.js
      'node_modules/zone.js/dist/zone.js',
      'node_modules/zone.js/dist/long-stack-trace-zone.js',
      'node_modules/zone.js/dist/proxy.js',
      'node_modules/zone.js/dist/sync-test.js',
      'node_modules/zone.js/dist/jasmine-patch.js',
      'node_modules/zone.js/dist/async-test.js',
      'node_modules/zone.js/dist/fake-async-test.js',

// RxJs
      { pattern: 'node_modules/rxjs/**/*.js', included: false,
      watched: false },      { pattern: 'node_modules/rxjs
      /**/*.js.map', included: false,        watched: false },

// Angular itself
      { pattern: 'node_modules/@angular/**/*.js', included:
      false,        watched: false },

{ pattern: 'systemjs.config.js', included: false, watched: false },
      { pattern: 'systemjs.config.extras.js', included: false,
      watched: false },
```

```
    'karma-test-shim.js',

        {pattern: 'app/**/*.js', included: false, watched: true}
]
```

Here, with the pattern, we have passed two options, `included` and `watched`. `included` refers to whether or not we want to include the file with the `<script>` tag; here, we will add it via SystemJS, so this is passed as `false`. And `watched` refers to whether or not this file will be watched during changes to the file. We have set `true`, as we want to watch the changes.

There seem to be a lot of files added, but these are basic and a must for running the test.

Let's take a closer look at what the files are. In the first chunk, these are mostly library files, including SystemJS as a module loader, zonejs as a sync handler, RxJS as a reactive library, and the Angular library itself.

Importantly, a new file in the second chunk is `karma-test-shim.js`, which we need to use with Karma as the module loader in test suites, that is, use SystemJS to load the modules in the Karma test runner. We will look at that file later in this section.

And then, it's all our app source file; we will put test/spec files on the same directory as well so that they will load all of our module files, including their test/spec files.

In addition to the files, we may need to change a few more configuration points based on requirements, as follows:

- `plugins`: This is required as Karma will use these `npm` modules to perform. If we plan to use more `npm` modules, we will need to add them here; for example, if we plan to use PhantomJS as our browser, we will need to add `'karma-phantomjs-launcher'` in the list:

```
plugins: [
    'karma-jasmine',
    'karma-chrome-launcher'
]
```

- `frameworks`: This doesn't need to change for now, as by default it chooses Jasmine; however, it will need to be updated if we plan to use other frameworks, such as Mocha. In that case, the following option should be updated:

```
frameworks: ['jasmine'],
```

- `browsers`: This is useful when we need to run the test in multiple browsers, and most of the time, we may need to run the test in PhantomJS, so we can add multiple browsers, as shown:

```
browsers: ['Chrome', 'PhantomJS']
```

So far, these are the basic changes we need in the `karma.con.js` files.

Let's take a look at our `karma.conf.js` file and see what it looks like:

```
module.exports = function(config) {

  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher')
    ],

    files: [
      // System.js for module loading
      'node_modules/systemjs/dist/system.src.js',

      // Polyfills
      'node_modules/core-js/client/shim.js',
      'node_modules/reflect-metadata/Reflect.js',

      // zone.js
      'node_modules/zone.js/dist/zone.js',
      'node_modules/zone.js/dist/long-stack-trace-zone.js',
      'node_modules/zone.js/dist/proxy.js',
      'node_modules/zone.js/dist/sync-test.js',
      'node_modules/zone.js/dist/jasmine-patch.js',
      'node_modules/zone.js/dist/async-test.js',
      'node_modules/zone.js/dist/fake-async-test.js',

      // RxJs
      { pattern: 'node_modules/rxjs/**/*.js', included: false,
      watched: false },
      { pattern: 'node_modules/rxjs/**/*.js.map', included:
      false,        watched: false },

      // Paths loaded via module imports:
      // Angular itself
      { pattern: 'node_modules/@angular/**/*.js', included:
      false,        watched: false },
```

```
      { pattern: 'node_modules/@angular/**/*.js.map', included:
      false, watched: false },
      { pattern: 'systemjs.config.js', included: false,
      watched:        false },
      { pattern: 'systemjs.config.extras.js', included: false,
      watched: false },

      'karma-test-shim.js',

      { pattern: 'app/**/*.js', included: false, watched: true }
    ],

    port: 9876,
    colors: true,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false
  })
}
```

Another important file that we added in the files list is `karma-test-shim.js`; and as we mentioned earlier, it's needed for us to use SystemJS (the module loader) with Karma. We have copied the file from the Angular quick start project and it's possible to change this based on the project structure.

Let's take a look at our `karma.conf.js` file:

```
Error.stackTraceLimit = 0; // "No stacktrace"" is usually best for app
testing.

jasmine.DEFAULT_TIMEOUT_INTERVAL = 1000;

var builtPath = '/base/app/';

__karma__.loaded = function () { };

function isJsFile(path) {
  return path.slice(-3) == '.js';
}

function isSpecFile(path) {
  return /\.spec\.(.*\.)?js$/.test(path);
}

function isBuiltFile(path) {
  return isJsFile(path) && (path.substr(0, builtPath.length) ==
  builtPath);
}
```

```
var allSpecFiles = Object.keys(window.__karma__.files)
  .filter(isSpecFile)
  .filter(isBuiltFile);

System.config({
  baseURL: 'base',
  // Extend usual application package list with test folder
  packages: { 'testing': { main: 'index.js', defaultExtension: 'js'
  } },

  // Assume npm: is set in `paths` in systemjs.config
  // Map the angular testing umd bundles
  map: {
    '@angular/core/testing':
'npm:@angular/core/bundles/core-testing.umd.js',
'@angular/common/testing':
'npm:@angular/common/bundles/common-testing.umd.js',
'@angular/compiler/testing':
'npm:@angular/compiler/bundles/compiler-testing.umd.js',
'@angular/platform-browser/testing':
'npm:@angular/platform-browser/bundles/
platform-browser-testing.umd.js',
'@angular/platform-browser-dynamic/testing':      'npm:@angular/platform-
browser-dynamic/bundles
 /platform-browser-dynamic-testing.umd.js',
'@angular/http/testing':
'npm:@angular/http/bundles/http-testing.umd.js',
'@angular/router/testing':
'npm:@angular/router/bundles/router-testing.umd.js',
'@angular/forms/testing':
'npm:@angular/forms/bundles/forms-testing.umd.js',
  },
});

System.import('systemjs.config.js')
  .then(importSystemJsExtras)
  .then(initTestBed)
  .then(initTesting);

/** Optional SystemJS configuration extras. Keep going w/o it */
function importSystemJsExtras(){
  return System.import('systemjs.config.extras.js')
  .catch(function(reason) {
    console.log(
      'Warning: System.import could not load the optional
"systemjs.config.extras.js". Did you omit it by accident?        Continuing
without it.'
    );
```

```
    console.log(reason);
  });
}

function initTestBed(){
  return Promise.all([
    System.import('@angular/core/testing'),
    System.import('@angular/platform-browser-dynamic/testing')
  ])

  .then(function (providers) {
    var coreTesting    = providers[0];
    var browserTesting = providers[1];

    coreTesting.TestBed.initTestEnvironment(
      browserTesting.BrowserDynamicTestingModule,
      browserTesting.platformBrowserDynamicTesting());
  })
}

// Import all spec files and start karma
function initTesting () {
  return Promise.all(
    allSpecFiles.map(function (moduleName) {
      return System.import(moduleName);
    })
  )
  .then(__karma__.start, __karma__.error);
}
```

# Testing the Karma runner

The initial set up of Karma is almost done; we will have to run our test and see how it's coming along. One more step before we run–we have to add the `karma` task in the `npm` script to run via the `npm` command. For that, we will have to add a task, as `test`, in the script section of the `package.json` file:

```
"scripts": {
     "test": "karma start karma.conf.js"
  }
```

After adding this snippet, we can run the test via `npm`, with `npm test`, which works the same way as `karma start`:

```
$ npm test
```

So, finally, we are ready to run our test via Karma. However, oops, we are getting some error! It's missing the `jasmine-core` module that is needed to run the test; in fact, there could be more modules missing.

The output with the error looks as follows:

```
> karma start karma.conf.js

module.js:328
    throw err;
    ^

Error: Cannot find module 'jasmine-core'
    at Function.Module._resolveFilename (module.js:326:15)
    at Function.require.resolve (internal/module.js:16:19)
```

Yes, we are indeed missing the modules and these are actually the development dependencies in our test suite. We will get to know a bit more about them in the following section.

## Missing dependencies

Though we are building a basic test suite for Angular, we are still missing a few required npm modules and these are the development dependencies for our test suite, which are as follows:

- `jasmine-core`: This states that Jasmine is our test framework
- `karma`: This is the Karma test runner of our test suite
- `karma-chrome-launcher`: This is required to launch Chrome from Karma, as we defined Chrome as our browser in `karma.config`
- `karma-jasmine`: This is the Karma adapter for Jasmine

As long as these are the dependencies, we should install these modules and include them in `package.json` as well.

We can install all of these together, as shown here:

```
$ npm install jasmine-core karma karma-chrome-launcher karma-
jasmine --save-dev
```

After successfully installing all the required dependencies, it seems like we are done with our configuration process, and we are ready to run `test` again:

```
$ npm test
```

The command output should state something like this:

```
$ Chrome 39.0.2623 (Mac OS X 10.10.5): Executed 0 of 0 ERROR
(0.003 secs / 0 secs)
```

That is it. Karma is now running for the first Angular application.

# Testing with Angular and Karma

The purpose of this first test using Karma is to create a dynamic to-do list. This walk-through will follow the TDD steps we discussed in `Chapter 1`, *Introduction to Test-Driven Development*: test first, make it run, and make it better. This will allow us to gain more experience in using TDD with an Angular application.

## A development to-do list

Before we start the test, let's set our focus on what needs to be developed using a development to-do list. This will allow us to organize our thoughts.

Here is the to-do list:

- **Maintain a list of items**: The example list consists of test, execute, and refactor
- **Add an item to the list**: The example list after we add the item is test, execute, refactor, and repeat
- **Remove an item from the list**: The example list after we add and remove the item is test, execute, and refactor

## Testing a list of items

The first development item is to provide us with the ability to have a list of items on a component. The next couple of steps will walk us through the TDD process of adding the first feature using the TDD life cycle.

# Test first

Determining where to start is often the most difficult part. The best way is to remember the three **A**s (**Assemble**, **Act**, and **Assert**) and start with the base Jasmine template format. The code to do this is as follows:

```
describe('Title of the test suite', () => {
    beforeEach(() => {
        // ....
    });
    it('Title of the test spec', () => {
        // ....
    });
});
```

Let's look at the explanation:

- `describe`: This defines the main feature that we are testing. The string will explain the feature in readable terms, and then the function will follow the test.
- `beforeEach`: This is the assemble step. The function defined in this step will be executed before every assert. It is best to put the required test setup before each test in this function.
- `it`: This is the act and assert step. In the `it` section, we will perform the action being tested, followed by some assertion. The act step doesn't have to go into the `it` function. Depending on the test, it might be more suited in the `beforeEach` function.

# The three As – Assemble, Act, and Assert

Now that the template is ready, we can start fitting in the pieces. We will again follow the three As mantra.

The following are the two parts of the assemble section.

In the first part, we initialize the component and execute the contractor of the class using the following code:

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

import { AppComponent } from './app.component';

beforeEach(async(() => {
   TestBed.configureTestingModule({
      declarations: [ AppComponent ]
```

```
      })
      .compileComponents();
  }));

beforeEach(() => {
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
});
...
```

Here, we imported some Angular testing APIs, such as `async` and `Testbed`. Here, `async` is used to load the required module to Bootstrap the application for the test suite, and `TestBed` is the primary API for writing a unit test for the Angular API. It has some service to create, compile, and initiate the components in the test suite.

We haven't defined the `AppComponent` component, but we will do this after we get a failing test.

The second part talks about the `AppComponent` object. The `AppComponent` object will contain the list of items on its `this` variable. Add the following code to `beforeEach` to get the component object:

```
// comp will hold the component object
let comp: AppComponent;
let fixture: ComponentFixture<AppComponent>;
beforeEach(() => {
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
});
```

In assert, there are two parts again:

The first assertion is to ensure that the `AppComponent` object has an `items` variable defined with three items. The `items` variable will be used to hold the list of all the items:

```
it('Should define a list object', () => {
        expect(comp.items).toBeDefined();
    });
```

The second and third assertions will be used to confirm whether the data in the list is correct:

```
//Second test
it('Should have 3 items in list', () => {
        expect(comp.items.length).toBe(3);
    });
```

```
//Third test
it('List items should be as expected', () => {
        expect(comp.items).toEqual(['test','execute','refactor']);
    });
```

That's it; the first is test, the second is execute, and the third is refactor.

## Make it run

The next step in the TDD life cycle is to make the application run and fix the code so that the tests pass. Remember, think about the smallest components that can be added to make the test pass by proceeding with the following steps:

1. Run the Karma test by typing the following command:

```
$ npm start
$ npm test
```

2. If we encounter the `TypeError: app_component_1.AppComponent is not a constructor` error, then it can be due to the following:
    - The preceding error message is saying that the `AppComponent` object hasn't been defined. Since the error message is telling us what is required, this is the perfect place to start.
3. Add the `AppComponent` class to the `app.component.ts` file, as follows:

```
export class AppComponent { };
```

4. Run the `start` and `test` command again from the `npm` console. We should now see a new error. **Error:** `The expected undefined to be defined as follow`
    - The new error message is again clear. We can also see that the code has now passed up to the point of our assertion at the following point:

```
expect(comp.items).toBeDefined();
```

    - As there are no items in the object, we need to add one. Update the `app/app.component.ts` file as follows:

```
export class AppComponent {
    items:Array<string>;
};
```

5. Let's run the `start` and `test` commands again from the `npm` console. We should now see one of the three tests pass! This means that we have successfully used TDD and Karma to get our first test to pass. Now, we need to fix the other three.
    - The next error is: `expected 0 to equal 3`
    - The error output again describes exactly what needs to happen. We just need to initialize the array with the elements test, execute, and run. Let's go to `app/app.component.ts` and add the data to the array initialization:

```
export class AppComponent {
    items:Array<string>;
    constructor() {
        this.items = ['test','execute','refactor'];
    }
};
```

6. Run the `start` and `test` commands again from the npm console. Excellent! The output is in green and states that all the tests have passed. The result component and class code from this step are as follows:

```
import {Component} from '@angular/core';

@Component({
    // ...
})

export class AppComponent {
    items:Array<string>;
    constructor() {
        this.items = ['test','execute','refactor'];
    }
};
```

Now that the *Make it run* step is complete, we can move on to the next step and make it better.

## Make it better

Until this point, there was nothing required to directly refactor or that had been identified in the development to-do list. A review of the development to-do list shows that an item can be crossed out:

- **View a list of to-do list items**: The example list consists of test, execute, and refactor
- **Add an item to a to-do-list**: The example list after we add the item will consist of test, execute, refactor, and new item

Next up is the requirement to add a new item to the list. The TDD rhythm will be followed again: test first, make it run, and make it better.

# Adding a function to the component class

The next task is to give the class the ability to add items to the object. This will require the addition of a method to the object. This walk-through will follow the same TDD steps that we followed previously.

## Test first

Instead of creating a new file and duplicating some of the assemble steps, the following test will be inserted under the last `it` method. The reason is that the same module and controller will be used:

```
describe('Title of the test suite', () => {
    let app:AppComponent;

    beforeEach(() => {
        // ....
    });
    it('Title of the test spec', () => {
        // ....
    });

    describe('Testing add method', () => {

    beforeEach(() => {
        // ....
    });
    it('Title of the test spec', () => {
        // ....
```

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

## The three As – Assemble, Act, and Assert

Now that the template is ready, we can start filling in the gaps using the 3 As mantra:

- **Assemble**: There is no initialization or setup required as the component and object will be inherited.
- **Act**: Here, we need to act on the `add` method with a new item. We place the `act` function into the `beforeEach` function. This allows us to repeat the same step if/when more tests are added:

```
beforeEach(() => {
     comp.add('new-item')
});
```

- **Assert**: Here, an item should be added to the list, and then you need to confirm that the last item in the array is as expected:

```
it('Should have 4 items in list', () => {
     expect(comp.items.length).toBe(4);
});
it('Should add a new item at the end of list', () => {
    var lastIndexOfList = comp.items.length - 1;
    expect(comp.items[lastIndexOfList]).toEqual('new-item');
});
```

## Make it run

The next step in the TDD life cycle is to make it run. Remember, think about the smallest components that can be added to make the test pass, as follows:

- Ensure that Karma is running in our console by typing in the following commands:

```
$ npm start
$ npm test
```

- The first error will state `TypeError: undefined is not a function`.

  This error refers to the following line of code:

  ```
  app.add('new-item');
  ```

  The error is telling us that the `add` method hasn't been defined. The `add` function will need to be added to the `app/app.component.ts` code. The class has already been defined, so the `add` function needs to be placed in the class:

  ```
  add() {
          this.items.push('new-item');
  };
  ```

  Note how the `add` function doesn't contain any logic. The smallest component has been added to get the test to satisfy the error message.

- Review the console window for the next error.

Success! All five tests have now passed.

The resulting code added to get the tests to pass is as follows:

```
import {Component} from '@angular/core';

@Component({
    selector: 'my-app',
    template: `<h3>MY Items</h3><ul><li *ngFor="let item of items">{{
    item }}</li></ul>`
})

export class AppComponent {
    items:Array<string>;
    constructor() {
        this.items = ['test','execute','refactor'];
    }
    add() {
        this.items.push('new-item');
    }
};
```

## Make it better

The main thing that we need to refactor is that the `add` function still hasn't been fully implemented. It contains a hard coded value, and the minute we send in a different item into the `add` function, the test will fail.

Keep Karma running so that we can keep passing the tests as changes are made. The main issue with the current `add` method is as follows:

- It doesn't accept any parameters
- It doesn't push a parameter onto the list but uses a hardcoded value

The resultant `add` function should now look as follows:

```
add(item) {
    this.items.push(item);
};
```

Run the `start` and `test` commands again from the `npm` console. Confirm that the Karma output still displays `SUCCESS`:

```
$ Chrome 49.0.2623 (Mac OS X 10.10.5): Executed 5 of 5 SUCCESS
(0.016 secs / 0.002 secs)
```

# Configuring Karma with Travis CI

**Continuous integration** (**CI**) is a development practice where developers need to integrate code into a shared repository. It is run in the automated build process with a test when any change happens in the codebase. This detects the error early, before it is pushed to production. There are lot of CI services around, including Travis CI, Jenkin CI, Circle CI, and so on.

In this section, we will see how we can integrate Karma with Travis.

## Travis CI

Travis CI is a popular hosted continuous integration platform that integrates with the GitHub project/repository to automatically run the test with every change in the code base of any branch or even with a pull request. It's easy to get the integration system just by putting a `.travis.yml` file in the project root with some configuration information about the project.

So, we may ask, why Travis? There are a few other CI services around. If we compare Travis to the other CI services, it has some benefits over the others:

- It's a hosted service; there is no need to host, install, and configure
- It's free and open source
- It has a separate test code for every branch, so it is easy to run a test for an individual branch

# Configuring Travis

As we said, we will have a `.travis.yml` file in our project directory with some configuration and information about our project.

Here is what the basic configuration in the YAML file looks like:

- **Specify the language**: We have used Node.js here:

```
language: node_js
node_js:
    -  "4"
```

- **Command or script**: This is required to run before or after each build; as shown here, this script will set the `git` username every time before running the build:

```
before_script:
    -  git config – –global user.name jquerygeek
```

In the preceding example, we have passed the configuration to run the build process in a real browser (Firefox) with a virtual screen, by default, with karma. It has run the process in the PhantomJS headless browser. This might come in handy, as long as Travis supports the real browser beside PhantomJS:

```
before_script:
    -  export DISPLAY=:99.0
    - sh –e /etc/init.d/xvfb start
```

- **Notifications**: This is required to set the notifications for e-mail and chat. Here, we have set `false` for `email`, as we do not want overtime e-mail notifications about the builds:

```
notifications:
    email: false
```

# Setting up the test with Karma

As seen earlier, we are guessing that we have the `package.json` file in our project root among the npm packages; if not, let's create a `package.json` file over there and add these configuration snips. Here, the first one is a dependency of Karma and the second one is to set the required params for the `npm test`, as Travis will run this to trigger our test. These will tell Travis how to run our test:

```
'devDependencies': {
    'karma': '~0.12'
},

'scripts': {
    'test': 'karma start  - -single-run - -browsers PhantomJS '
}
```

Our initial setup and configuration are ready for testing. We defined the Karma dependency, as Travis will run `nmp install` for every suite, and will take the necessary steps for adding Karma. And for running the test, it will call the `npm test`, and we defined how that test task will run the test. Here, we have set the default browser to PhantomJS so that the test will run with it. However, if we need to run the test with a different browser, we should define that in the `.travis.yml` file with the `before_script` command, as we showed earlier for Firefox.

In that case, `npm test` will not run the test with the custom browser; for that, we have to do a custom call with the browser name, as follows:

```
karma start - -browsers Firefox - -single-run
```

# Self-test questions

The following self-test questions will help you further test your knowledge of using TDD with AngularJS and Karma:

Q1. How do you use Karma to create a configuration file?

1. `karma config`
2. `karma init`
3. `karma -config karma.conf.js`

Q2. The Jasmine test method, named `before`, gets executed before every test.

1. True
2. False

Q3. Bower is used to install Karma.

1. True
2. False

Q4. The 3 As stand for which one of these?

1. A group of super heroes
2. Assemble, Act, and Assert
3. Accept, Approve, and Act

# Summary

In this chapter, we discussed how Karma became important. We saw how to install, configure, and run Karma. Finally, we walked through an Angular example project using Karma with TDD.

In the next chapter, we will learn about end-to-end testing with Protractor.

# 4
# End-to-End Testing with Protractor

Unit testing is only one aspect of testing that only tests the responsibility of every piece of code. However, when it comes to testing the flow and feature of any component, module, or full application, then **end-to-end** (**e2e**) testing is the only solution.

In this chapter, we will look at the end-to-end testing flow step by step through all the layers of an application. We will be introduced to Protractor, the end-to-end testing tool from the Angular team. We already know the reason behind it, why it was created, and what kind of problems it solves.

In this chapter, we will go through:

- The process of installing and configuring Protractor
- Implementing the Protractor end-to-end test in our existing Angular project
- e2e test runs
- Returning the result of the test

## An overview of Protractor

Protractor is an end-to-end testing tool that runs using Node.js and is available as an npm package. Before talking about Protractor specifically, we need to understand what end-to-end testing is.

We already learned briefly about end-to-end testing in `Chapter 2`, *Details of JavaScript Testing*. But let's have a quick recap:

End-to-end testing is testing an application against all the interconnected moving parts and layers of an application. This differs from unit tests, where the focus is on individual components, such as classes, services, and directives. With end-to-end testing, the focus is on how the application or a module, as a whole, works, such as confirming that the click of a button triggers x, y, and z actions.

Protractor allows the end-to-end testing of any module or even any size of web application by interacting the DOM elements of that application. It offers the ability to select a specific DOM element, share the data with that element, simulate the click of a button, and interact with an application in the same way as a user would. It then allows expectations to be set based on what the user would expect.

# Core of Protractor

In a quick overview, we got a basic idea about Protractor–how it needs to select DOM elements and interact with them just like a real user would in order to run an e2e test on any application. To do these activities, Protractor provides some global functions; some are from its core API, and some are from WebDriver. We will discuss them in detail in `Chapter 5`, *Protractor, a Step Ahead*.

However, let's take a quick overview now:

- **Browser**: Protractor provides the global function `browser`, which is a global object from WebDriver that is mostly used to interact with the application browser where the application is running during the e2e test process. It provides some useful methods to interact with, as follows:

```
browser.get('http://localhost:3000'); // to navigate the
browser to a specific url address
browser.getTitle(); // this will return the page title that
defined in the projects landing page
```

  And there are many more, which we will discuss in the next chapter.

- **Element**: This is a global function provided by Protractor; it's basically used to find a single element based on the locator, but it supports multiple element selection as well, by chaining another method, `.all` as `element.all`, which also takes `Locator` and returns `ElementFinderArray`. Let's have a look at an `element` example:

```
element(Locator); // return the ElementFinder
element.all(Locator); // return the ElementFinderArray
element.all(Locator).get(position);  // will return the
defined  position
element from the ElementFinderArray
element.all(Locator).count(); // will return the
total number in the select element's array
```

And there are many more, which we will discuss in the next chapter.

- **Action**: As we have seen, the `element` method will return a selected DOM `element` object, but we need to interact with a DOM and the actions for doing that job come with some built-in methods. The DOM will not contact the browser unit with any action method calls.Let's have a look at few of action's example:

```
element(Locator).getText(); // return the ElementFinder
based on locator
element.(Locator).click(); // Will trigger the click
handler for that specific element
element.(Locator).clear(); // Clear the field's value
(suppose the element is input field)
```

And there are many more, which we will discuss in the next chapter.

- **Locator**: This actually informs Protractor how to find a certain element in the DOM element. Protractor exports `Locator` as a global factory function, which will be used with a global `by` object.Let's have a look at a few examples of `Locator`:

```
element(by.css(cssSelector)); // select element by css
selector
element(by.id(id)); //  select element by element ID
element.(by.model); // select element by ng-model
```

And there are many more, which we will discuss in the next chapter.

# A quick example

Now we can go through a quick example considering the following user specification.

Assuming that I input `abc` into the search box, the following should occur:

- The search button should be clicked on
- At least one result should be received

The preceding specification describes a basic search feature. Nothing in the preceding specification describes a controller, directive, or service; it only describes the expected application behavior. If a user were to test the specification, they may perform the following steps:

1. Point the browser to the website.
2. Select the input field.
3. Type `abc` in the input field.
4. Click on the **Search** button.
5. Confirm that the search output displays at least one result.

The structure and syntax of Protractor mirrors that of Jasmine and the tests we have written in `Chapter 3`, *The Karma Way*. We can think of Protractor as a wrapper around Jasmine, with added features to support end-to-end testing. To write an end-to-end test with Protractor, we can follow the same steps that we just saw, but with the code.

Here are the steps with code:

1. Point the browser to the website:

```
browser.get('/');
```

2. Select the input field:

```
var inputField = element.all(by.css('input'));
```

3. Type `abc` in the input field:

```
inputField.setText('abc');
```

4. Click on the **Search** button:

```
var searchButton = element.all(by.css('#searchButton');
searchButton.click();
```

5. Find the search result details on the page:

```
var searchResults = element.all(by.css('#searchResult');
```

6. Finally, the assertion needs to be made that at least one or more search results are available on the screen:

```
expect(searchResults).count() >= 1);
```

As a complete test, the code will be as follows:

```
describe('Given I input 'abc' into the search box',function(){
    //1 - Point browser to website
    browser.get('/');
    //2 - Select input field
    var inputField = element.all(by.css('input'));
    //3 - Type abc into input field
    inputField.setText('abc');
    //4 - Push search button
    var searchButton = element.all(by.css('#searchButton');
    searchButton.click();
    it('should display search results',function(){
    // 5 - Find the search result details
    var searchResults = element.all(by.css('#searchResult');
    //6 - Assert
    expect(searchResults).count() >= 1);
    });
});
```

That's it! When Protractor runs, it will open a browser, go to the website, follow the instructions, and finally check the expectations. The trick with end-to-end testing is having a clear vision of what the user specification is and then translating that specification to code.

The previous example is a high-level view of what will be described throughout this chapter. Now that we have been introduced to Protractor, the rest of the chapter will show how Protractor works behind the scenes, how to install it, and finally, walk us through a complete example using TDD.

# Origins of Protractor

Protractor is not the first end-to-end testing tool that the Angular team built. The first tool was called **Scenario Runner**. In order to understand why Protractor was built, we need to first look at its predecessor–Scenario Runner.

Scenario Runner is in the maintenance mode and has reached the end of its life. It has been deprecated in the place of Protractor. In this section, we will look at what Scenario Runner was and what gaps the tool had.

# The birth of Protractor

Julie Ralph is the primary contributor of Protractor. According to Julie Ralph, the motivation for Protractor was based on the following experience with the Angular Scenario Runner on another project within Google (`http://javascriptjabber.com/106-jsj-protractor-with-julie-ralph/`):

> *"We tried using the Scenario Runner. And we found that it really just couldn't do the things that we needed to test. We needed to test things like logging in. Your login page isn't an Angular page, and the Scenario Runner couldn't deal with that. And it couldn't deal with things like popups and multiple windows, navigating the browser history, stuff like that."*

Based on her experience with the Scenario Runner, Julie Ralph decided to create Protractor to fill the gaps.

Protractor takes advantage of the maturity of the Selenium project, and wraps up its methods so that it can be easily used for Angular projects. Remember, Protractor is about testing through the eyes of the user. It was designed to test all the layers of an application: web UI, backend services, persistence layer, and so on.

# Life without Protractor

Unit testing is not the only testing that needs to be written and maintained. Unit tests focus on small individual components of an application. By testing small components, the confidence in the code and logic grows. Unit tests don't focus on how the complete system works when interconnected.

End-to-end testing with Protractor allows the developer to focus on the complete behavior of a feature or module. Going back to the search example, the test should only pass if the whole user specification passes; enter data into the search box, click on the **Search** button, and see the results. Protractor is not the only end-to-end testing framework out there, but it is the best choice for Angular applications. Here are a few reasons why you should choose Protractor:

- It is documented throughout the Angular tutorials and examples
- It can be written using multiple JavaScript testing frameworks, including Jasmine and Mocha
- It provides convenience methods for Angular components, including waiting for a page to load, expectations on promises, and so on
- It wraps Selenium methods that automatically wait for promises to be fulfilled

- It is supported by **SaaS** (**Software as a Service**) providers, such as Sauce Labs, which is available at `https://saucelabs.com/`
- It is supported and maintained by the same company that maintains Angular and Google

# Getting ready with Protractor

It's time to start getting our hands dirty and install and configure Protractor. Installations and applications are constantly changing. The main focus will be on the specific configuration used in this book, and not an in-depth installation guide. There are several different configurations, so review the Protractor site for additional details. To find the latest installation and configuration guide, visit `http://angular.github.io/protractor/`.

## Installation prerequisites

Protractor has the following prerequisites:

- **Node.js**: Protractor is a Node.js module available using npm. The best way to install Node.js is to follow the instructions on the official site at `http://nodejs.org/download/`.
- **Chrome**: This is a web browser built by Google. It will be used to run end-to-end tests in Protractor without the need for a Selenium server. Follow the installation instructions on the official site at `http://www.google.com/chrome/browser/`.
- **Selenium WebDriver for Chrome**: This is a tool that allows you to interact with web applications. Selenium WebDriver is provided with the Protractor `npm` module. We will walk through the instructions as we install Protractor.

# Installing Protractor

Here are the steps to install Protractor:

1. Once Node.js is installed and available in the Command Prompt, type the following command to install Protractor in the current directory:

```
$ npm install protractor
```

2. The preceding command uses Node's `npm` command to install Protractor in the current local directory.
3. To use Protractor in the Command Prompt, use the relative path to the Protractor bin directory.
4. Test that the Protractor version can be determined as follows:

```
$ ./node_modules/protractor/bin/protractor --version
```

# Installing WebDriver for Chrome

Here are the steps to install WebDriver for Chrome:

1. To install Selenium WebDriver for Chrome, go to the `webdriver-manager` executable in the Protractor `bin` directory that can be found at `./node_modules/protractor/bin/`, and type the following:

```
$ ./node_modules/protractor/bin/webdriver-manager update
```

2. Confirm the directory structure.
3. The preceding command will create a Selenium directory containing the required Chrome driver used in the project.

The installation is now complete. Both Protractor and Selenium WebDriver for Chrome have been installed. We can now move on to the configuration.

# Customizing configuration

In this section, we will be configuring Protractor using the following steps:

1. Start with a standard template configuration.
2. Fortunately, the Protractor installation comes with some base configurations in its installation directory.

3. The one that we will use is called `conf.js` located in the `protractor/example` section.
4. Review the example configuration file:

   The `capabilities` parameter should only specify the name of the browser:

```
exports.config = {
//...
capabilities: {
  'browserName': 'chrome'
},
//...
};
```

   The framework parameter should specify the testing framework name, and we will use Jasmine here:

```
exports.config = {
//...
framework: 'jasmine'
//...
};
```

   The final important configuration is the source file declaration:

```
exports.config = {
  //...
  specs: ['example_spec.js'],
  //...
};
```

Excellent! Now we have Protractor installed and configured.

# Confirming the installation and configuration

To confirm installation, Protractor requires at least one file defined in the `specs` configuration section. Before adding a real test and complicating things, create an empty file called `confirmConfigTest.js` in the root directory. Then, edit the `conf.js` file located in the project root and add the test file to the `specs` section so that it looks as follows:

```
specs: ['confirmConfigTest.js'],
```

To confirm that Protractor has been installed, run Protractor by going to the root of our project directory and type the following:

```
$ ./node_modules/protractor/bin/protractor conf.js
```

If everything was set up correctly and installed, we will see something similar to this in our Command Prompt:

```
Finished in 0.0002 seconds
0 tests, 0 assertions, 0 failures
```

# Common installation and configuration issues

The following are some common issues that you might come across while installing WebDriver for Chrome:

| Issues | Solution |
|---|---|
| Selenium not installed correctly | If the tests have errors related to the Selenium WebDriver location, you need to ensure that you followed the steps to update WebDriver. The update step downloads the WebDriver components into the local Protractor installation folder. Until WebDriver has been updated, you won't be able to reference it in the Protractor configuration. An easy way to confirm the update is to look in the Protractor directory and ensure that a Selenium folder exists. |
| Unable to find tests | When no tests are executed by Protractor, it can be frustrating. The best place to start is in the configuration file. Ensure that the relative paths and any filenames or extensions are correct. |

For a complete list, refer to the official Protractor site at `http://angular.github.io/protractor/`.

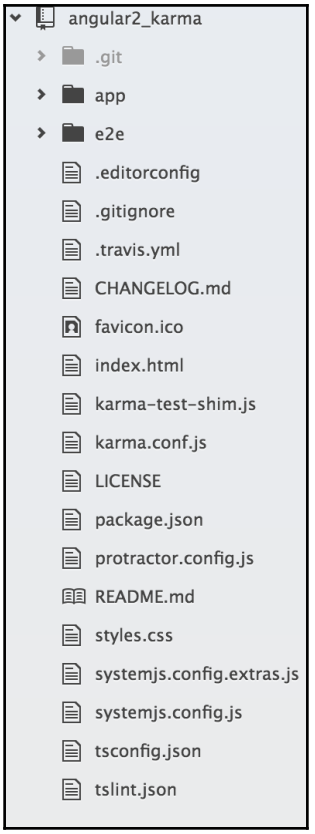
# Integrating Protractor with Angular

So far, we have seen how to install and configure Protractor and we also have a basic overview of how Protractor works. In this section, we will go through with the process of integrating Protractor in an existing Angular project, where we just have unit testing and will see how Protractor is used in e2e testing in real life.

# Getting the existing project

The code in this test will leverage the unit tested code from `Chapter 3`, *The Karma Way*. We will copy the code to a new directory called `angular-protractor`.

As a reminder, the application is a to-do application that has some items in the to-do list; let's add some more items to the list. It has a single component class, `AppComponent`, that has a list of items and an `add` method. The current code directory should be structured as follows:



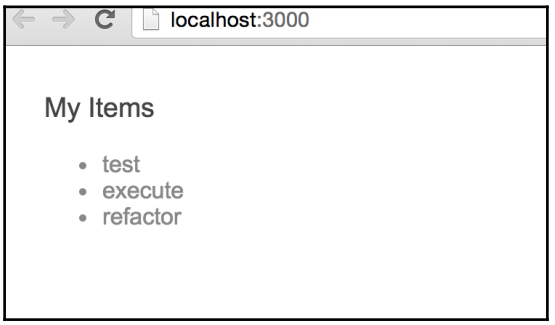

After getting this structure, the first job is to get the required dependencies, `node_modules` locally by running the following:

```
$ npm install
```

This will install all the required modules; next, let's build and run the project with the `npm` command:

```
$ npm start
```

Everything should be fine; the project should run on `http://localhost:3000` and the output should be as follows:




And yes, we are ready to go to the next step to implement Protractor in our Angular project.

# The Protractor setup flow

The setup will mirror the installation and configuration steps that we saw earlier in this chapter:

1. Install Protractor.
2. Update Selenium WebDriver.
3. Configure Protractor based on the example configuration.

We will follow the Protractor installation and configuration steps we have covered in the previous section in a new project directory. The only difference is that the Protractor tests could be named with e2e prefix, such as `**.e2e.js`. This will allow us to easily identify the Protractor tests in our project structure.


This absolutely depends on a developer's choice; some people just put the Protractor test in a new directory with a subfolder, `spec/e2e`. It's just part of structuring the project.

# Installing Protractor

We might have Protractor set up globally or we might not, so it's always great to have Protractor installed in the project. And so, we will have Protractor installed locally and will add in `package.json` as `devDependency`.

To install Protractor in our project, run this command from the project directory:

```
$ npm install protractor –save-dev
```

We can check Protractor as follows:

```
$ ./node_modules/protractor/bin/protractor --version
```

This should provide the latest version, 4.0.10, as follows:

```
Version 4.0.10
```

**A good practice that we will follow**
We showed how to set up Protractor in a directory, but its good to have Protractor globally installed with the following command:

```
$ npm install –g protractor
```

This is so that we can use it easily to call Protractor from the command shell as with `protractor`; to know the Protractor version, we can call it as follows:

```
$ protractor –version
```

# Updating WebDriver

To update Selenium WebDriver, go to the `webdriver-manager` executable in the Protractor `bin` directory that can be found at `./node_modules/protractor/bin/` and type the following:

```
$ ./node_modules/protractor/bin/webdriver-manager update
```

A good practice to follow, as informed, we will install Protractor globally, and if so, we will have the `webdriver-manager` command global as well, so that, we can easily run it for `update`, as shown here:

```
$ webdriver-manager update
```

This will update WebDriver and will support the latest browsers.

# Getting ready

As we have cloned the sample quick start project, it has already integrated and configured Protractor in the project. For the purpose of learning, we would like to integrate Protractor in the existing project.

To do so, we will have to remove the existing `protractor.config.js` file from the project root directory.

# Setting up the core configuration

As we have seen earlier, Protractor configurations will be stored in a JS file. We will need to create a configuration file in our project root; let's name it as `protractor.config.js`.

For the time being, keep the changeable fields empty, as these are on the project structure and configuration dependent. So, the initial look could be something like this and these configuration options are known to us:

```
exports.config = {

    baseUrl: ' ',

    framework: 'jasmine',

    specs: [],

    capabilities: {
        'browserName': 'chrome'
    }

};
```

As long as our project will run locally on port `3000`, our `baseUrl` variable will be as follows:

```
exports.config = {
    // ....
    baseUrl: ' http://localhost:3000',
    // ....
};
```

We are planning to keep our e2e test spec in the same folder where we put the unit test files, `app/app.component.spec.ts`. This time it will have a new e2e prefix and will look like `app/app.component.e2e.ts`. Based on that, our specs and config will be updated:

```
exports.config = {
    // ....
    specs: [
        'app/**/*.e2e.js'
    ],
    // .....
};
```

As long as it's an Angular project, we need to pass an extra configuration, `useAllAngular2AppRoots: true`, as it will tell Protractor to wait for all the Angular apps' root elements on the page instead of just the one root element matching:

```
exports.config = {
    // ....
    useAllAngular2AppRoots: true,
    // .....
};
```

We are running our project via the node server; so, we need to pass one more configuration option so that Jasmine itself supports node. This configuration is a must to pass in the Jasmine 2.x version, but we may not need it if we use Jasmine 1.x. Here, we have added the two most common options in `jasmineNodeOpts`; there are a few that are used based on requirements:

```
exports.config = {
    // ....
    jasmineNodeOpts: {
        showColors: true,
        defaultTimeoutInterval: 30000
    },
    // .....
};
```

# Diving into the test specifics

To run the Protractor test, we need two files: one is the configuration file that we already created in our project root as `protractor.conf.js`, and the next one is the spec, where we will define the e2e test spec, and that one will be located in the app folder as `app/app.component.e2e.ts`.

So, let's see the file that we should define there:

```
describe('Title for test suite', () => {

    beforeEach(() => {
        // ...
    });

    it('Title for test spec', () => {
        // ...
    });

});;
```

These syntaxes should be known to us as we already used the Jasmine syntax in our unit test suite as well.

Let's have a quick recap

- `describe`: This contains the block of codes to run the test suite
- `beforeEach`: This is used to contain the setup code, which is used in every test spec
- `it`: This is used to define the test spec and contain the specific code for that test spec to run

The main part of running an e2e test for any website is to get the DOM elements of that website and then interact with those elements through the test process. And so, we will need to get the DOM elements of our running project.

As long as the current project is running in the web browser, we will have to get the instance of the browser itself first; interestingly, Protractor provides that with the global browser object. With this browser object, we can get all browser-level commands, such as `browser.get`, and we can navigate through to our project URL:

```
beforeEach(() => {
    browser.get('');
});;
```

With this `browser.get('')` method, we will navigate to the root of our project.

We have the global browser object, with which we can get the title of the running page, which is basically, the title we defined here in the project `index.html` file. `browser.getTitle` will provide the title and then we can match that as expected. So, our test spec will look like this:

```
it('Browser should have a defined title', () => {
       expect(browser.getTitle()).toEqual('Angular Protractor');
});
```

If we take a quick look, our short e2e test spec will look as follows:

```
describe('AppComponent Tests', () => {
    beforeEach(() => {
        browser.get('');
    });

    it('Browser should have a defined title', () => {
        expect(browser.getTitle()).toEqual('Angular Protractor');
    });
});
```

Time to run the e2e test with Protractor. The command will look as follows:

```
$ protractor protractor.conf.js
```

The result is as expected–0 failure as we have set the `index.html` page title to **Angular Protractor.** The result will be as follows:

```
1 spec, 0 failures
Finished in 1.95 seconds
```

Time to move on and add a new test spec for the rest of the DOM elements of the page, where we have list items listed on the page; so, we will test them automatically via Protractor.

At first, we will check whether we have all the three items listed. We have already learned in an earlier section of this chapter about some Protractor common APIs, but a quick recap, we will use the `element.all` method by passing some locator (`by.css`, `by.id`, and `by.model`) to get the elements array object. And then, we can use the Jasmine matcher to match with the expected value, as shown:

```
it('Should get the number of items as defined in item object', () => {
        var todoListItems = element.all(by.css('li'));
        expect(todoListItems.count()).toBe(3);
    });
```

We should get the result passed, as we have three items listed in the UI.

We can add a few more test specs for testing the UI elements. For example, to check whether the items listed are in the correct order, we can check their label, as follows:

```
it('Should get the first item text as defined', () => {
        expect(todoListItems.first().getText()).toEqual('test');
    });

    it('Should get the last item text as defined', () => {
        expect(todoListItems.last().getText()).toEqual('refactor');
    });
```

We have matched the first and last item's label/text with the expected value and it should pass as well.

Let's combine all the test specs in the e2e file. It will look like this:

```
describe('AppComponent Tests', () => {
    var todoListItems = element.all(by.css('li'));

    beforeEach(() => {
        browser.get('/');
    });

    it('Browser should have a defined title', () => {
        expect(browser.getTitle()).toEqual('Angular Protractor');
    });

    it('Should get the number of items as defined in item object', ()
    => {
        expect(todoListItems.count()).toBe(3);
    });

    it('Should get the first item text as defined', () => {
        expect(todoListItems.first().getText()).toEqual('test');
    });

    it('Should get the last item text as defined', () => {
        expect(todoListItems.last().getText()).toEqual('refactor');
    });
});
```

Let's run all the specs together:

```
$ protractor protractor.conf.js
```

As expected, all the tests should pass and the result will be as follows:

```
4 specs, 0 failures
Finished in 2.991 seconds
```

As long as we named our Protractor config file `protractor.conf.js`, we don't need to mention the configuration file name while running it with the `protractor` command; Protractor will get its configuration file by itself. With any other name, we should mention the configuration file name with Protractor.

So, in this case, we can just run the test as follows:

```
$ protractor
```

The result will be the same as it was before.

# Running the test via NPM

In this project, we will build and run a project via npm. In `Chapter 3`, *The Karma Way*, we ran the `karma` test via `npm`; similarly, we will run the `protractor` test with `npm` as well. And to do so, we have to add `protractor` in the `scripts` section in `package.json` of our project:

```
"scripts": {
    // ...
    "e2e": "protractor"
    // ....
  };
```

To install `protractor` in our project, run from the project directory:

```
$ npm e2e
```

In some OSes, this command may produce some `npm` error. This is actually for `webdriver-manager`, which may not updated. To solve this we have to add the `webdriver-manager` update script to `npm` and run it only once at the very first time, like this:

```
"scripts": {
    // ...
    "webdriver-update": "webdriver-manager update"
    // ....
  };
```

We also have to run it as:

```
$ npm webdriver-update
```

That's it, we are ready to run the e2e test again and this should work exactly the same as the `protractor` command.

Let's confirm this:

```
$ npm run e2e
```

The expected result will be as follows:

```
4 specs, 0 failures
Finished in 2.991 seconds
```

# Making the test better

There are a couple of things that were discussed in this chapter that need further clarification. These include the following:

- Where is the asynchronous logic?
- How do we really implement TDD with end-to-end tests?

# Async magic

In the preceding tests, we saw some magic that you might be questioning. Here are some of the magic components that we glanced over:

- Loading a page before test execution
- Assertion on elements that get loaded in promises

## Loading a page before test execution

In the previous test, we used the following code to specify that the browser should point to the home page:

```
browser.get('');
```

The preceding command will launch the browser and navigate to the `baseUrl` location. Once the browser reaches the page, it will have to load Angular and then implement the Angular-specific functions. Our tests don't have any wait logic, and this is part of the beauty of Protractor with Angular. The waiting for page loading is already built into the framework for us. Our tests can then be written very cleanly.

## Assertion on elements that get loaded in promises

The assertions and expectations already have promise fulfillment written in them. In the case of our test, we wrote the assertion so that it expects the count to be `3`:

```
expect(todoListItems.count()).toBe(3);
```

However, in reality, we may have thought that we needed to add asynchronous testing to the assertion in order to wait for the promise to be fulfilled, involving something more complicated like the following:

```
it('Should get the number of items as defined in item object', (done) => {
    var todoListItems = element.all(by.css('li'));
    todoListItems.count().then(function(count){
        expect(count).toBe(3);
        done();
    });
});
```

The preceding code is longer, more granular, and more difficult to read. Protractor has the ability to make tests more concise for certain elements built in to expectations.

# TDD with Protractor

With our first test, there is a clear distinction of end-to-end tests and unit tests. With the unit test, we focused on strong coupling the test to the code. As an example, our unit test spied on the scope for a specific component class, `AppComponent`. We have to initiate the component to get the instance of the component, as follows:

```
import {AppComponent} from "./app.component";

beforeEach(() => {
    app = new AppComponent();
});
```

In the Protractor test, we don't care about which component class we are testing and our focus is on the user perspective of the test. We start with the selection of a particular element within the DOM; in our case, that element is tied to Angular. The assertion is that the number of elements for a specific repeater is equal to the expected count.

With the loose coupling of the end-to-end test, we can write a test that focuses on the user specification, which initially displays three elements, and then have the freedom to write that in the page, class, component, and so on, in the way we want.

# Self-test questions

Use TDD with Protractor to develop the third development to-do list item.

Q1. Which of the following frameworks does Protractor use?

- Selenium
- Unobtanium
- Karma

Q2. You can install Protractor with any existing Angular project.

- True
- False

Q3. Karma and Protractor can run together in a single project.

- True
- False

Q4. Which team has developed Protractor?

- ReactJS team
- Angular team
- NodeJS team

# Summary

This chapter gave us an overview of end-to-end testing with Protractor and also provided the necessary idea to install, configure, and apply end-to-end testing with an existing Angular project. Protractor is an important part of testing any Angular application. It bridges the gap to ensure that a user's specifications work as expected. When end-to-end tests are written according to user specifications, the confidence of the application and ability to refactor grows. In the upcoming chapters, we will see how to apply Karma and Protractor in more depth with simple and straightforward examples.

The next chapter will walk us through more details on Protractor with some advance configuration, details on some APIs, and it will also debug the tests.

# 5
# Protractor, a Step Ahead

End-to-end testing is real fun as long as it interacts directly with the browser, but a good developer should know the advanced features of Protractor to perform large-scale application testing. Besides that, debugging is kind of a challenge in e2e testing as it depends on the DOM element of the browser.

Protractor has some APIs for debugging. This chapter will mostly cover those APIs and features, including the following:

- Setting up and configuring Protractor
- Some advanced Protractor APIs such as browser, locator, and action
- Debugging Protractor with the `browser.pause()` and `browser.debug()` APIs

## Advanced setup and configuration

In the previous chapter, we saw a basic and commonly used setup and configuration for Protractor. Here, we will take a look at some advanced configurations that make installation simpler and more powerful.

# Installing Protractor globally

Here are the steps to install Protractor globally:

1. Once Node.js has been installed and is available in command prompt, type the following command to install Protractor globally on the system:

```
$ npm install -g protractor
```

   The preceding command uses Node's `npm` command to install Protractor globally so that we can use Protractor just with the `protractor` command.

2. Test whether the Protractor version can be determined as follows:

```
$ protractor --version
```

# Advanced configuration

In this section, we will be configuring Protractor a bit more using the following steps:

1. Update the protractor `config` file to support multiple browsers in a single test suite. The `multiCapabilities` parameter is an array that takes multiple `browserName` objects for any test suite, as shown here:

```
exports.config = {
  //...
multiCapabilities: [{
 'browserName': 'firefox'
}, {
 'browserName': 'chrome'
}]
//... };
```

2. We can set advanced settings for browsers in the `capabilities` parameter; for example, for `chrome`, we can pass extra parameters as `chromeOptions`, as follows:

```
exports.config = {
  //...
  capabilities: {
    'browserName': 'chrome'
    'chromeOptions': {
      'args': ['show-fps-counter=true']
    }}]
```

```
//... };
```

3. Sometimes, we may need to run Protractor directly without Selenium or WebDriver. This is possible by passing a parameter in the `config.js` file. The parameter is `directConnect: true` in the configuration object, as shown here:

```
exports.config = {
  //...
  directConnect: true,
  //...
};
```

Great! We have configured Protractor a step ahead.

# Protractor APIs

The main activities of an e2e test for any web page are to get the DOM elements of that page, interact with them, assign an action to them, and share information with them; then, the user can get the current state of the website. To enable us to perform all these actions, Protractor provides a wide array of APIs (some are from the web driver). In this chapter, we will look at some commonly used APIs.

In the previous chapter, we saw how Protractor works with an Angular project, where we had to interact with UI elements. For that, we used a few Protractor APIs, such as `element.all`, `by.css`, `first`, `last`, and `getText`. However, we didn't see or understand the workings of these APIs in depth. To understand the workings of APIs in Protractor is very simple, but in real life we will mostly have to work with bigger, complex projects. Hence, it's important that we understand and know more about these APIs in order to interact with the UI and play with its events.

## Browser

Protractor works with Selenium WebDriver, which is a browser automation framework. We can use a method from the Selenium WebDriver API to interact with the browser from the test spec. We will take a look at a few of them in the following sections.

To navigate the browser to a specific web address and load the mock modules for that page before the Angular load, we will use the `.get()` method by passing the specific address or relative path:

```
browser.get(url);
browser.get('http://localhost:3000'); // This will navigate to
the localhost:3000 and will load mock module if needed
```

To get the current page's web URL, use the `CurrentUrl()` method, as shown here:

```
browser.getCurrentUrl(); // will return http://localhost:3000
```

To navigate to another page and browse it using in-page navigation, `setLocation` is used, as follows:

```
browser.setLocation('new-page'); // will change the url and navigate to the
new url, as our current url was http://localhost:3000, now it will change
and navigate to http://locahost:3000/#/new-page
```

To get the title of the current page (basically, the title that is set in the HTML page), the `getTitle` method is used, as shown here:

```
browser.getTitle(); // will return the page title of our page, for us it
will return us "Angular Protractor Debug"
```

To reload the current page with the mocks module before the Angular load, the `refresh()` method is used, as follows:

```
browser.refresh(); // this will reload the full page and definitely will
load the mocks module as well.
```

To pause the test process, the `pause()` method is used. This is useful for debugging the test process, and we will use this test debugging section:

```
browser.pause();
```

To debug the test process, the `debugger()` method is used. This method is different and can be considered an advanced level of the `pause()` method. This is useful for advanced debugging of the test process, along with injecting custom helper function into the browser. We will use this test debugging section as well:

```
browser.debugger();
```

To close the current browser, `close()` is used. This is useful for complex multimodule testing, when we sometimes need to close the current browser before opening a new one:

```
browser.close();
```

To support Angular in Protractor, we have to set the `useAllAngularAppRoots` params to `true`. The logic behind doing this is that, when we set this parameter to `true`, it will search for all Angular apps in the page while the element finder traverses through the page:

```
browser.useAllAngular2AppRoots;
```

# Elements

Protractor itself exposes some global functions, and `element` is one of them. This function takes a locator (a kind of selector–we will discuss it in the next step) and returns an `ElementFinder`. This function basically finds a single element based on the locator, but it supports multiple element singing along with chaining another method as `element.all`, which also takes a locator and returns an `ElementFinderArray`. Both of them support chaining methods for the next action.

## element.all

As we already know, `element.all` returns an `ElementArrayFinder` that supports chaining methods for the next action. We will look at a few of them and how they actually work:

To select multiple elements as an array with a specific locator, we should use `element.all`, as follows:

```
element.all(Locator);
var elementArr = element.all(by.css('.selector'));  // return the
ElementFinderArray
```

After getting a bunch of elements as an array, we may need to select a specific element. In that case, we should be chaining `get(position)` by passing the specific array index as the position number:

```
element.all(Locator).get(position);
elementArr.get(0); // will return first element from the ElementFinderArray
```

After getting a bunch of elements as an array, we might need to select child elements again with a preferred locator, and for that we can chain the `.all(locator)` method again with the existing elements, as shown here:

```
element.all(Locator).all(Locator);
elementArr.all(by.css('.childSelector')); // will return another
```

```
ElementFinderArray as child elements based on child locator
```

After getting the desired elements, we might want to check whether the number of elements selected is as expected. There is a method, `count()`, that is used to chain to get the total number of selected elements:

```
element.all(Locator).count();
elementArr.count(); // will return the total number in the select element's
array
```

Similar to the `get(position)` method, we can get the first element from the array by chaining the `first()` method:

```
element.all(Locator).first();
elementArr.first(); // will return the first element from the element's
array
```

Similar to the `first()` method, we can get the last element from the array by chaining the `last()` method:

```
element.all(Locator).last();
elementArr.last(); // will return the last element from the element array
```

As long as we have a bunch of elements as an array, we may need to traverse though the elements to take any action. In that case, we may need to go through a loop by chaining the `each()` method:

```
element.all(Locator).each(Function) { };
elementArr.each( function (element, index) {
    // ......
}); // ... will loop through out the array elements
```

Just like the `each()` method, there is another method, `filter()`, to chain with the element array to traverse through the items and assign a filter to them:

```
element.all(Locator).filter(Function) { };
elementArr.filter( function (element, index) {
    // ......
}); //... will apply filter function's action to all elements
```

## element

The `element` class returns `ElementFinder`, which means a single element in the element array, and this also supports chaining methods for the next action. In the previous examples, we saw how to obtain a single selected element from the element array so that all of the chaining methods work on that single element as well. There are a lot of chaining methods for working on a single element, and we will look at a few that are most commonly used.

By passing a specific locator as an argument to the `element` method, we can select a single DOM element, as shown here:

```
element(Locator);
var elementObj = element(by.css('.selector'));  // return the ElementFinder
based on locator
```

After getting a specific single element, we may need to find the child element of the element on which we have to chain the `element.all` method with the rerun `elementFinder` object. For this, pass a specific locator to find the child `elementFinderArray`, as follows:

```
element(Locator).element.all(Locator);
elementObj.element.all(by.css('.childSelector')); // will return another
ElementFinderArray as child elements based on child locator
```

After selecting a specific element, we might need to check whether that element is present while chaining the `isPresent()` method, as follows:

```
element(Locator).isPresent();
elementObj.isPresent(); // will return boolean if the selected element is
exist or not.
```

# Actions

Actions mainly change the method that affects or triggers the selected DOM element. The goal of selecting a DOM element is to interact with it by triggerring some actions so that it can act like a real user. There are some commonly used actions for specific interaction. We will look at a few of them here.

To get the inner text or contained text of any element, we have to chain the `getText()` method with the `elementFinder` object after selecting the specific element, as follows:

```
element(Locator).getText();
var elementObj = element(by.css('.selector'));  // return the ElementFinder
```

```
based on locator
elementObj.getText(); // will return the contained text of that specific
selected element
```

To get the inner HTML of any element, we have to chain the `getInnerHtml()` method with the `elementFinder` object after selecting the specific element, as shown here:

```
element.(Locator).getInnerHtml();
elementObj.getInnerHtml(); // will return the inner html of the selected
element.
```

We can find any specific attribute value of any element by passing the attribute key to the `getAttribute()` method, which will chain with the selected `elementFinder` object, as follows:

```
element(Locator).getAttribute('attribute');
elementObj.getAttribute('data'); // will return the value of data attribute
of that selected element if that have that attribute
```

In most cases, we need to clear the value of the input field. For that, we can chain the `clear()` method with the selected `elementFinder` object, as shown:

```
element.(Locator).clear();
elementObj.clear(); // Guessing the elementFinder is input/textarea, and
after calling this clear() it will clear the value and reset it.
```


Remember that it's only the input or texture that may have some value and needs you to clear/reset the value.

When we need to trigger a click event on any button, link, or image, after selecting a specific `elementFinder` object, we will need to chain the `click()` method, and it will act like a real click on that element:

```
element.(Locator).click();
elementObj.click(); // will trigger the click event as the selected element
chaining it.
```

Sometimes, we might need to trigger the `submit()` method for form submission. In that case, we have to chain the `submit()` method with the selected element. The selected element should be a `form` element:

```
element.(Locator).submit();
elementObj.submit(); // Will trigger the submit for the form
element as submit() work only for form element.
```

# Locators

Locators inform Protractor how to find a certain element in the DOM element. Protractor exports `locator` as a global factory function, which will be used with a global `by` object. We can use them in many ways based on our DOM, but let's look at some of the most commonly used ones.

We can select any element by passing any of the CSS selectors to the `by.css` method, as shown here:

```
element(by.css(cssSelector));
element.all(by.css(cssSelector));
<span class="selector"></span>
element.all(by.css('.selector'));  // return the specific DOM
element/elements that will have selector class on it
```

We can select any element by passing its element ID to the `by.id` method, as shown here:

```
element(by.id(id));
<span id="selectorID"></span>
element(by.id('selectorID')); // return the specific DOM element that will
have selectorID as element id on it
```

We can also select a specific element or elements by tag name by passing it to `by.tagName`, as follows:

```
element(by.tagName(htmlTagName));
element.all(by.tagName(htmlTagName));
<span data="myData">Content</span>
element.all(by.tagName('span')); // will return the DOM element/elements of
all span tag.
```

To select the DOM element of any specific input field, we can pass the name in the `by.name` method, as shown:

```
element(by.name(elementName));
<input type="text" name="myInput">
element(by.name('myInput')); // will return the specific input field's DOM
element that have name attr as myInput
```

Besides a CSS selector or ID, we can select a specific button by passing its text label to `by.buttonText`:

```
<button name="myButton">Click Me</button>
element(by.buttonText('Click Me')); // will return the specific button that
will have Click Me as label text
element(by.buttonText(textLabel));
```

We can find an element by passing the model name defined as an `ng-model` on `by.model`, as shown here:

```
element.(by.model);
<span ng-model="userName"></span>
element(by.model('userName')); // will return that specific element which
have defined userName as model name
```

Similarly, we can find a specific DOM element by passing its binding defined with `ng-bind` in `by.bindings`, as follows:

```
element.(by.binding);
<span ng-bind="email"></span>
element(by.binding('email')); // will return the element that have email as
bindings with ng-bind
```

Besides all the locators explained earlier, there is another way to find a specific DOM element: a custom locator. Here, we have to create a custom locator using `by.addLocator` by passing the locator name and callback. Then, we have to pass that custom locator with `by.customLocatorName(args)`, as shown here:

```
element.(by.locatorName(args));
<button ng-click="someAction()">Click Me</button>
by.addLocator('customLocator', function(args) {
    // .....
})
element(by. customLocator(args)); // will return the element that will
match with the defined logic in the custom locator. This useful mostly when
user need to select dynamic generated element.
```

# Protractor tests – postmortem

It's kind of difficult to debug e2e tests as they depend on the entire ecosystem of an application. Sometimes they depend on prior actions such as login, and sometimes they depend on permissions. Another major barrier to debugging e2e is its dependency on WebDriver. As it acts differently with different operating systems and browsers, this makes it difficult to debug e2e. Besides that, it generates long error messages, which makes it difficult to separate browser related issues and test process errors.

Still, we will try to debug all e2e tests and see how that works for our case.

# Types of failure

There might be various reasons for the failure of a test suite as long as it depends on WebDriver and various parts throughout the system.

Let's look at some known failure types:

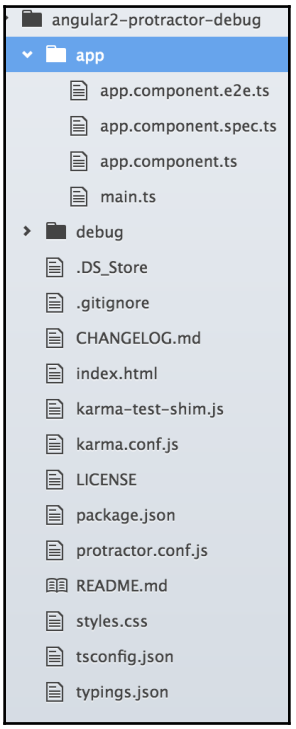
- **WebDrive failure**: WebDriver throws an error when a command can't be completed. For example, a browser can't get the address that's defined to help it navigate, or maybe an element is not found as expected.
- **WebDriver unexpected failure**: Sometimes, WebDriver fails and gives an error when it fails to update the web driver manager. This is a browser and OS-related issue, although it's not common.
- **Protractor failure for Angular**: Protractor will fail when Angular is not found in the library as expected because the Protractor test depends on Angular itself.
- **Protractor Angular2 failure**: Protractor will fail for an Angular project's test spec when the `useAllAngular2AppRoots` parameter is missing in the configuration because, without this, the test process will look at one single root element while expecting more elements in the process.
- **Protractor failure for timeout**: Sometimes, Protractor fails for the timeout when the test spec falls into a loop or a long pool and fails to return data in time. However, a timeout is configurable, so it can be increased as needed.
- **Expectation failure**: This is a common failure in the test spec.

# Loading an existing project

The code used in this test comes from `Chapter 4`, *End to End Testing with Protractor*. We will copy the code to a new directory: `angular-protractor-debug`.

As a reminder, the application was a to-do application that had some items in the to-do list, and we added some items to it. It has a single component class, `AppComponent`, which has a list of items, and an `add` method.

The current directory should be structured as follows:



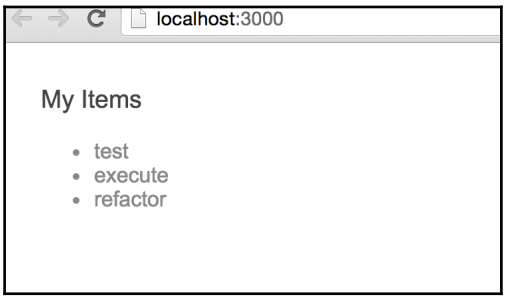

After verifying that the folder structure is the same as shown in the preceding screenshot, the first job is to get the required dependencies, `node_modules`, locally by running the following:

```
$ npm install
```

This will install all the required modules. Now, let's build and run the project with the `npm` command:

```
$ npm start
```

Everything should be fine now: the project should run on `http://localhost:3000`, and the output should be as shown:




With that, we are ready to go on to the next step of implementing a debugger in our Angular project.

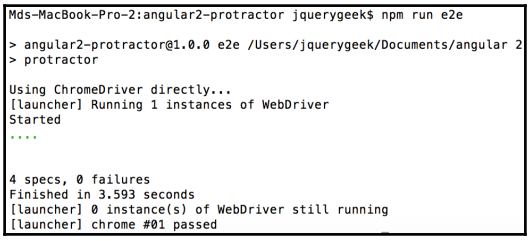
# Including the debugger in the project

Before adding the debugger to our project, let's run the e2e test in our existing project. We hope to pass without any failure in the e2e test specs.

Let's run it using the following command:

```
$ npm run e2e
```

As expected, our tests passed. The result is as follows:




We can add our debugging code in the same place where the passed test specs are, but let's keep the passed test case separate and play with the debugger in a different directory. Let's create a new directory, `debug/`. We will need two files in the directory: one for the configuration and the other for the spec.

For the Protractor configuration file, we can copy the `protractor.conf.js` file and rename it to `debugConf.js`.

Everything in the configuration will be the same as the previous configuration. However, we need to increase the default timeout for the Jasmine test, or the test will timeout during debugging.

Let's increase the timeout to `3000000` ms:

```
exports.config = {
    // ....
    jasmineNodeOpts: {
      showColors: true,
      defaultTimeoutInterval: 3000000
    },
    // .....
};
```

Next, we will need a spec file to which to write the test spec and debug tests. Save the new spec file as `app.debug.e2e.ts`. Oh yes, we will need to change the config file once again to define the spec files for debugging.

```
exports.config = {
    // ....
    specs: [
      'app.debug.e2e.js'
    ],
    // .....
};
```

We can make a simple test spec file for `app.debug.e2e.ts`. Then, we can add debug codes and play with them.

Simple test specs are as shown:

```
describe('AppComponent Tests', () => {
    beforeEach(() => {
        browser.get('/');
    });

    it('Test spec for debug and play', () => {
    });
});
```

# Pausing and debugging

To debug any test spec, we must pause the test process and see what is going on step-by-step. Protractor also has built-in methods to pause the process. Here are two methods to pause and debug the test process:

- `browser.pause()`
- `browser.debugger()`

# Using pause

Debugging Protractor tests becomes easy and simple with the `browser.pause()` command. Using the `pause()` method, we can enter the Protractor debugger control flow and execute some commands to check out what's happening in the control flow of the test. Mostly, developers use debuggers in tests when the test fails with an unknown error and there are long error messages.

After using the `browser.pause()` command, we can use a few more commands based on our needs.

Let's take a brief look:

- `c`: If we type `c` as a command, it will move one step forward in the test, and we will see how the test command works in depth. If we plan to move forward with a test, it's better to do it quickly as there is a timeout issue ( the Jasmine default timeout), which we've learned about. We will see an example later on.
- `repl`: By typing `repl` as a command, we can enter the interactive mode of debugging. It's known as as the interactive mode as, from there, we can interact directly with the browser from the terminal by typing WebDriver commands. A response, result, or error from the browser will be displayed on the terminal as well. We will see more hands-on examples later.
- `Ctrl + C`: Press *Ctrl* + C to exit pause mode and continue the test. When we will use this, the test will continue from the point where it paused.

# A quick example

To use `browser.pause()` in test specifications, we will have to add the method to the test spec at the places we want to pause the test and watch the control flow for debugging. Here, we just have one test spec with a wrong/failing test case, we know it will fail, and we will find out why it failed.

We will have to add the `pause()` method, as shown, to the test `spec it() {}` function:

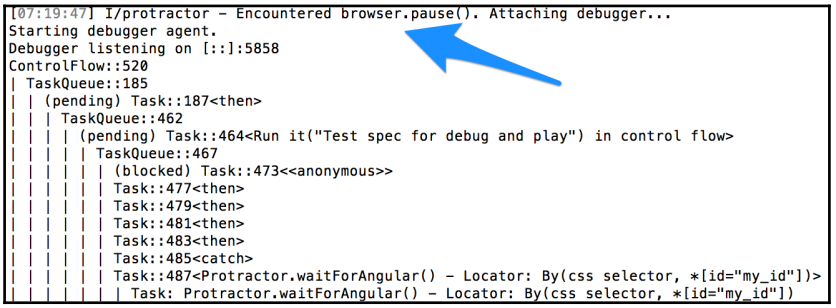
```
it('Test spec for debug and play', () => {
  browser.pause();
  // There is not element with the id="my_id", so this will fail
  the test
  expect(element(by.id('my_id')).getText()).toEqual('my text')
});
```

It's time to run the test. As we've separated the test spec for the debugger, we will run the test via Protractor (not `npm`).

Let's run the test with the following command:

```
$ protractor debug/debugConf.js
```

As we have placed the `browser.pause()` method before the `expect()` method, it will pause there. We can see in the control flow that this makes it wait for Angular:



We will move forward; for that, let's type `c`. It will run `executeAsyncScript` and will wait for Angular to load:

```
-- Next command: executeAsyncScript
ControlFlow::520
| TaskQueue::185
| | (pending) Task::187<then>
| | | TaskQueue::462
| | | | (pending) Task::464<Run it("Test spec for debug and play") in control flow>
| | | | | TaskQueue::467
| | | | | | (pending) Task::487<Protractor.waitForAngular() - Locator: By(css selector, *[id="my_id"])>
| | | | | | | TaskQueue::599
| | | | | | | | (pending) Task::601<then>
| | | | | | | | | (active) TaskQueue::604
| | | | | | | | Task::603<then>
| | | | | | Task::489<then>
| | | | | | Task::491<then>
| | | | | | Task::493<then>
| | | | | | Task::495<then>
| | | | | | Task::497<then>
| | | | | | Task::499<then>
| | | | | | Task::502<then>
| | | | | | Task::504<then>
| | | | Task::466<then>
```

We will move another step ahead by typing `c`. It will try to select the element based on the locator provided by us, that is, `element(by.id('my_id'))`:

```
ControlFlow::520
| TaskQueue::185
| | (pending) Task::187<then>
| | | TaskQueue::462
| | | | (pending) Task::464<Run it("Test spec for debug and play") in control flow>
| | | | | TaskQueue::467
| | | | | | (pending) Task::493<then>
| | | | | | | TaskQueue::669
| | | | | | | | (pending) Task::678<WebDriver.findElements(By(css selector, *[id="my_id"]))>
| | | | | | | | | TaskQueue::689
| | | | | | | | | | (pending) Task::691<then>
| | | | | | | | | | | (active) TaskQueue::694
| | | | | | | | | | Task::693<then>
| | | | | | | | Task::680<catch>
| | | | | | Task::495<then>
| | | | | | Task::497<then>
| | | | | | Task::499<then>
| | | | | | Task::502<then>
| | | | | | Task::504<then>
| | | | Task::466<then>
```

We are close to getting the test result now. For that, we will have to move another step forward by typing `C`. Now, it will try to select the element based on the locator, and it will fail to select that. This will give a result with an error message, as expected:

```
Failures:
1) AppComponent Tests Test spec for debug and play
  Message:
    Failed: No element found using locator: By(css selector, *[id="my_id"])
  Stack:
    NoSuchElementError: No element found using locator: By(css selector, *[id="my_id"])
        at WebDriverError (/usr/local/lib/node_modules/protractor/node_modules/selenium-webdriver/lib/error.js:27:5)
        at NoSuchElementError (/usr/local/lib/node_modules/protractor/node_modules/selenium-webdriver/lib/error.js:242:5)
        at /usr/local/lib/node_modules/protractor/built/element.js:719:27
        at ManagedPromise.invokeCallback_ (/usr/local/lib/node_modules/protractor/node_modules/selenium-webdriver/lib/promise.js:1379:14)
        at TaskQueue.execute_ (/usr/local/lib/node_modules/protractor/node_modules/selenium-webdriver/lib/promise.js:2913:14)
        at TaskQueue.executeNext_ (/usr/local/lib/node_modules/protractor/node_modules/selenium-webdriver/lib/promise.js:2896:21)
        at asyncRun (/usr/local/lib/node_modules/protractor/node_modules/selenium-webdriver/lib/promise.js:2775:27)
        at /usr/local/lib/node_modules/protractor/node_modules/selenium-webdriver/lib/promise.js:639:7
        at process._tickDomainCallback (internal/process/next_tick.js:129:7)Error
```

## Debugging with interactive mode

To enter interactive mode, we have to type `repl`, after which we can run any command used in the test spec.

Let's find the element and its text:

```
> element(by.id('my_id')).getText()
```

The result is the same as we got previously, by moving forward step by step, by typing `C`.

**Result**: `NoSuchElementError: No element found using locator: By (css selector,*[id="my_id"])`

Now, let's see how interactive mode works for a valid locator, when the `element`will be found:

```
> element.all(by.css('li')).first().getText()
```

**Result**: `test`

# Using the debugger

Debugging with the `browser.debugger()` command is a bit complex and more advanced than using `browser.pause()`. With the `browser.pause()` command, we can pause the control flow of the test and inject the custom helper functions into the browser so that debugging happens in the same way as we would debug in a browser console.

This debugging should be under node in debug mode, like here in Protractor debugging.

This debugging is not useful for someone bad at node debugging.

Here is an example:

To use the `browser.debugger()` method in the test spec, we will have to add the method to the test spec at the points where we want to set a breakpoint and watch the control flow.

For us, we have to add the `debugger()` method, as follows, to the `test spec it() {}` function, and this will be our breakpoint:

```
it('Test spec for debug and play', () => {
  browser.debugger();
  // There is not element with the id="my_id", so this will fail
the test
  expect(element(by.id('my_id')).getText()).toEqual('my text')
  });
```

Now let's run it:

```
$ protractor debug debug/debugConf.js
```

To run the debugger, we have to add `debug` to the command after `protractor`.

After running the command, we have to move forward by typing `c`, but here we need to do it just once. The output is as follows:

```
[08:50:42] I/direct - Using ChromeDriver directly...
[08:50:42] I/launcher - Running 1 instances of WebDriver
Starting debugger agent.
Debugger listening on [::]:5858
connecting to localhost:5858 ... ok
c
Started
break in /usr/local/lib/node_modules/protractor/built/browser.js:664
 662         this.driver.executeScript(clientSideScripts.installInBrowser);
 663         webdriver.promise.controlFlow().execute(function () {
>664             debugger;
 665         }, 'add breakpoint to control flow');
 666     };
```

# Self-test questions

Q1. `Selenium WebDriver` is a browser automation framework.

- True
- False

Q2. Using `browser.debugger()` is a simple way to debug Protractor.

- True
- False

Q3. What are `by.css()`, `by.id()`, and `by.buttonText()` called?

- Elements
- Locators
- Actions
- Browsers

# Summary

Protractor has various kinds of API. In this chapter, we tried to understand some of the most commonly used APIs with some examples. We also covered API types (such as browser, elements, locator, and actions), and how they are chained with one another, in some detail.

Debugging was introduced in this chapter, and we tried to learn a simple way to debug, using `browser.pause()`, in more detail, and then we moved on to a complex method (`browser.debugger()`), and understood that complex developers need node debugger experience.

In the next chapter, we will delve into more real-life projects; further, we will go through the top-down and bottom-up approaches and learn them both.

# 6
# The First Step

The first step is always the most difficult. This chapter provides an initial introductory walk-through of how to use TDD to build an Angular application with a component, class, and model. We will be able to begin the TDD journey and see the fundamentals in action. Up to this point, this book has focused on a foundation of TDD and the tools required for it. Now, we will switch gear and dive into TDD with Angular.

This chapter will be the first step of TDD. We have already seen how to install Karma and Protractor, in addition to small examples and a walk-through on how to apply them. In this chapter, we will focus on:

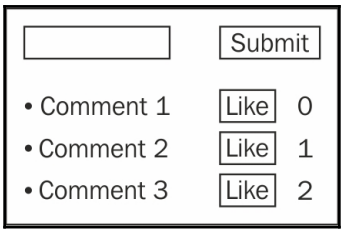
- Creating a simple comments application
- Integrating Karma and Protractor with the application
- Covering the components of testing and its associated classes

## Preparing the application's specifications

Create an application to enter comments. The specifications of the application are as follows:

- If I am posting a new comment, when I click on the **Submit** button, the comment should be added to the comment list
- For a comment, when I click on the **Like** button, the number of likes for the comment should increase

Now that we have the specifications of the application, we can create our development to-do list. It won't be easy to create an entire to-do list of the whole application. Based on the user specifications, we have an idea of what needs to be developed. Here is a rough sketch of the UI:




Hold back from jumping into the implementation and thinking about how we will use a component class, `*ngFor`, and so on. Resist, resist, resist! Although we can think of how this will be developed in the future, it is never clear until we delve into the code, and that is where we will start getting into trouble. TDD and its principles are here to help us get our mind and focus in the right place.

# Setting up the Angular project

In previous chapters, we discussed in detail how a project should be set up, looked at the different components involved, and walked through the entire process of testing. We will skip these details and provide a list in the following section for the initial actions to get the project set up and ready with a test configuration for unit and end-to-end testing.

# Loading an existing project

We will get a simple Angular project from the Angular team's examples and modify it for our implementation.

We will clone the `quickstart` project from the Angular GitHub repo and start with that one. We should have Git installed globally besides `node`/`npm`.

```
$ git clone https://github.com/angular/quickstart.git
angular-project
```

This will copy the project locally as `angular-project`; this project may contain some extra files (they may be updated continuously), but we will try to keep our project folder structure looking like this:


We will keep it simple initially, and then, we will add our required files step by step. This will make us more confident.

Let's proceed with it and run the following:

```
$ cd angular-project
$ npm install
```

The `npm install` command will install the required modules for project dependencies that are defined in the `package.json` file that is in the project's root.

# Setting up the directory

In previous examples, we put the component, unit test spec, and e2e test spec in the same folder just to keep it simpler. For a larger project, it's difficult to manage all in the same folder.

To make that manageable, we will put the test spec in a separate folder. Here, our sample `quickstart` project has already put test specs in the default folder, but we will have a new structure and will put our test files in the new structure.

Let's start setting up the project directory:

1. Navigate to the project's root folder:

```
cd angular-project
```

2. Initialize the test (`spec`) directory:

```
mkdir spec
```

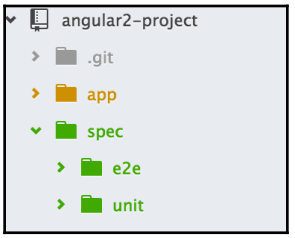
3. Initialize the `unit` test directory:

```
mkdir spec/unit
```

4. Initialize the end-to-end (`e2e`) test directory:

```
mkdir spec/e2e
```

Once the initialization is complete, our folder structure should look as follows:



# Setting up Karma

The details for Karma can be found in `Chapter 3`, *The Karma Way*. Here, we will mostly take a look at the Karma configuration file.

In this `quickstart` project, we have already installed and configured Karma, and we have the `karma.conf.js` file in the project directory.

To confirm that we have Karma in the system, let's install it globally using the following command:

```
npm install -g karma
```

As mentioned before, we already have Karma configured in this project as part of a `quickstart` project, and we have the `karma.conf.js` file in the project directory.

Now we will look at some basic configuration options that everyone should know. In this configuration file, there are some advanced options, such as test reporting and bug reporting. We will skip those as they are not very important to know at this initial stage.

Let's know more about a few of the configurations that we will need to proceed further.

The `basePath` should be updated when we have a custom path for the project in the server. For now, it's `''`, as this project is running in the root path. The next option is the `frameworks`; by default, we are using `jasmine` here, but we can change the framework name if we want to use others, such as `mocha`. One point to remember is that if we plan for a different framework, we will have to add the relevant plugin.

```
basePath: '',
    frameworks: ['jasmine'],
```

Plugins are required, as Karma will use these `npm` modules inside to perform the actions; for example, if we plan to use PhantomJS as the browser, we will need to add `'karma-phantomjs-launcher'` to the list:

```
plugins: [
        'karma-jasmine',
        'karma-chrome-launcher'
]
```

The next and most important option is `files[]`; with this, Karma will include all the required files for testing. It loads a file based on dependency. We will have all the required files in the `files[]` array.

First, we will add `System.js` since we use `systemjs` as the module loader in the app. Then, add `polyfills` for shim support on all browsers, `zone.js` for Async support in the app, RxJS as the reactive library, Angular library files, shim for Karma test, the components file, and, finally, the test specs. There might be some other files in the list for debug and reporting; we are skipping their explanations.

This is what our `files[]` array will look like:

```
files: [
      // System.js for module loading
      'node_modules/systemjs/dist/system.src.js',

      // Polyfills
      'node_modules/core-js/client/shim.js',
      'node_modules/reflect-metadata/Reflect.js',

      // zone.js
      'node_modules/zone.js/dist/zone.js',
      'node_modules/zone.js/dist/long-stack-trace-zone.js',
      'node_modules/zone.js/dist/proxy.js',
      'node_modules/zone.js/dist/sync-test.js',
      'node_modules/zone.js/dist/jasmine-patch.js',
      'node_modules/zone.js/dist/async-test.js',
      'node_modules/zone.js/dist/fake-async-test.js',

      // RxJs
      { pattern: 'node_modules/rxjs/**/*.js', included: false,
      watched: false },
      { pattern: 'node_modules/rxjs/**/*.js.map', included:
      false, watched: false },

      // Paths loaded via module imports:
      // Angular itself
      { pattern: 'node_modules/@angular/**/*.js', included:
      false, watched: false },
      { pattern: 'node_modules/@angular/**/*.js.map', included:
      false, watched: false },

      { pattern: 'systemjs.config.js', included: false, watched:
      false },
      { pattern: 'systemjs.config.extras.js', included: false,
      watched: false },
      'karma-test-shim.js',

      // transpiled application & spec code paths loaded via
      module imports
      { pattern: appBase + '**/*.js', included: false, watched:
```

```
            true },
            { pattern: testBase + '**/*.spec.js', included: false,
            watched: true },
],
```

That's all we need to know for now in the `karma.conf` file. We will proceed by updating these settings if needed.

Let's take a look at the complete `karma.conf.js` file:

```
module.exports = function(config) {

  var appBase    = 'app/';       // transpiled app JS and map files
  var appSrcBase = 'app/';       // app source TS files
  var appAssets  = 'base/app/'; // component assets fetched by
  Angular's compiler

  var testBase    = 'spec/unit/';       // transpiled test JS and map
  files
  var testSrcBase = 'spec/unit/';       // test source TS files

  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    plugins: [
      require('karma-jasmine'),
      require('karma-chrome-launcher'),
      require('karma-jasmine-html-reporter'), // click "Debug" in
      browser to see it
      require('karma-htmlfile-reporter') // crashing w/ strange
      socket error
    ],

    customLaunchers: {
      // From the CLI. Not used here but interesting
      // chrome setup for travis CI using chromium
      Chrome_travis_ci: {
        base: 'Chrome',
        flags: ['--no-sandbox']
      }
    },
    files: [
      // System.js for module loading
      'node_modules/systemjs/dist/system.src.js',

      // Polyfills
      'node_modules/core-js/client/shim.js',
      'node_modules/reflect-metadata/Reflect.js',
```

```
// zone.js
'node_modules/zone.js/dist/zone.js',
'node_modules/zone.js/dist/long-stack-trace-zone.js',
'node_modules/zone.js/dist/proxy.js',
'node_modules/zone.js/dist/sync-test.js',
'node_modules/zone.js/dist/jasmine-patch.js',
'node_modules/zone.js/dist/async-test.js',
'node_modules/zone.js/dist/fake-async-test.js',

// RxJs
{ pattern: 'node_modules/rxjs/**/*.js', included: false,
watched: false },
{ pattern: 'node_modules/rxjs/**/*.js.map', included: false,
watched: false },

// Paths loaded via module imports:
// Angular itself
{ pattern: 'node_modules/@angular/**/*.js', included: false,
watched: false },
{ pattern: 'node_modules/@angular/**/*.js.map', included:
false, watched: false },

{ pattern: 'systemjs.config.js', included: false, watched:
false },
{ pattern: 'systemjs.config.extras.js', included: false,
watched: false },
'karma-test-shim.js',

// transpiled application & spec code paths loaded via module
imports
{ pattern: appBase + '**/*.js', included: false, watched: true
},
{ pattern: testBase + '**/*.spec.js', included: false, watched:
true },

// Asset (HTML & CSS) paths loaded via Angular's component
compiler
// (these paths need to be rewritten, see proxies section)
{ pattern: appBase + '**/*.html', included: false, watched: true
},
{ pattern: appBase + '**/*.css', included: false, watched: true
},

// Paths for debugging with source maps in dev tools
{ pattern: appSrcBase + '**/*.ts', included: false, watched:
false },
{ pattern: appBase + '**/*.js.map', included: false, watched:
false },
```

```
        { pattern: testSrcBase + '**/*.ts', included: false, watched:
        false },
        { pattern: testBase + '**/*.js.map', included: false, watched:
        false }
      ],

      // Proxied base paths for loading assets
       proxies: {
         // required for component assets fetched by Angular's compiler
         "/app/": appAssets
      },

      exclude: [],
      preprocessors: {},
      // disabled HtmlReporter; suddenly crashing w/ strange socket
      error
      reporters: ['progress', 'kjhtml'],//'html'],

      // HtmlReporter configuration
      htmlReporter: {
        // Open this file to see results in browser
        outputFile: '_test-output/tests.html',

        // Optional
        pageTitle: 'Unit Tests',
        subPageTitle: __dirname
      },

      port: 9876,
      colors: true,
      logLevel: config.LOG_INFO,
      autoWatch: true,
      browsers: ['Chrome'],
      singleRun: true
    })
  };
```

# Test directory updated

We have seen details of `karma-test-shim.js` in `Chapter 3`, *The Karma Way*. This is needed to run unit testing via Karma.

We have changed the test specs directory/location, and `karma-test-shim.js` is configured based on the default structure of the project. Because we have moved the test to a different location and out of the `app/` folder, we need to update `karma-test-shim.js` accordingly.

Here is the change that will need to be made:

```
var builtPath = '/base/';
```

# Setting up Protractor

In `Chapter 4`, *End-to-End Testing with Protractor*, we discussed the full installation and setup of Protractor. This sample application already has Protractor installed and configured. So, we will just take a look at the `protractor.conf.js` file.

This configured Protractor instance has test reporting implemented. We will skip those parts from the config file here and just have a look at the common setting options.

Before we go to the config file overview, to be sure, we will install Protractor globally on the system:

```
$ npm install -g protractor
```

Update Selenium WebDriver:

```
$ webdriver-manager update
```

We will have to make sure that Selenium is installed.

As expected, `protractor.conf.js` is located at the root of the application. Here is the complete configuration of the `protractor.conf.js` file:

```
var fs = require('fs');
var path = require('canonical-path');
var _ = require('lodash');

exports.config = {
  directConnect: true,

  // Capabilities to be passed to the webdriver instance.
  capabilities: {
    'browserName': 'chrome'
  },

  // Framework to use. Jasmine is recommended.
```

```
    framework: 'jasmine',

    // Spec patterns are relative to this config file
    specs: ['**/*e2e-spec.js' ],


    // For angular tests
    useAllAngular2AppRoots: true,

    // Base URL for application server
    baseUrl: 'http://localhost:8080',

    // doesn't seem to work.
    // resultJsonOutputFile: "foo.json",

    onPrepare: function() {
      //// SpecReporter
      //var SpecReporter = require('jasmine-spec-reporter');
      //jasmine.getEnv().addReporter(new
      SpecReporter({displayStacktrace: 'none'}));
      //// jasmine.getEnv().addReporter(new SpecReporter({
      displayStacktrace: 'all'}));

      // debugging
      // console.log('browser.params:' +
      JSON.stringify(browser.params));
      jasmine.getEnv().addReporter(new Reporter( browser.params )) ;

      // Allow changing bootstrap mode to NG1 for upgrade tests
      global.setProtractorToNg1Mode = function() {
        browser.useAllAngular2AppRoots = false;
        browser.rootEl = 'body';
      };
    },

    jasmineNodeOpts: {
      // defaultTimeoutInterval: 60000,
      defaultTimeoutInterval: 10000,
      showTiming: true,
      print: function() {}
    }
  };
```

# Top-down versus bottom-up approach – which one do we use?

From a development perspective, we have to determine where to start. The approaches that we will discuss in this book are as follows:

- **The bottom-up approach**: With this approach, we think about the different components we will need (class, service, module, and so on) and then pick the most logical one and start coding.
- **The top-down approach**: With this approach, we work from the user scenario and UI. We then create the application around the components in the application.

There are merits to both these approaches, and the choice can be based on your team, the existing components, requirements, and so on. In most cases, it is best for you to make the choice based on the least resistance.

In this chapter, the approach of the specification is top-down; everything is laid out for you from the user scenario and will allow you to organically build the application around the UI.

# Testing a component

Before getting into the specifications and the mindset of the feature being delivered, it is important to understand the fundamentals of testing a component class. A component in Angular is a key feature used in most applications.

## Getting ready to go

Our sample application (`quickstart`) has some very basic test specs for unit and end-to-end testing. We will start the TDD approach from the very beginning, so we will not use any of the test specs and the existing component's code in our implementation.

For that, what can we do is just clean up this sample application, and we will just keep the folder structure and application bootstrap files.

So, first of all, we will have to remove the unit test file (`app.component.spec.ts`) and end-to-end test files (`app.e2e-spec.ts`). These are two test specs that existed in the application structure.

# Setting up a simple component test

When testing a component, it's important to inject the component into the test suite and then initiate the component class as the second task. The tests confirm that either the objects or methods in the component's scope are available as expected.

To have the component instance in the test suite, we will use the simple `import` statement in the test suite and initiate the component object in the `beforeEach` method so that we have a new instance of the component object for every test spec with that test suite. Here is an example of what this will look like:

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

import {AppComponent} from "../../app.component";

describe('AppComponent Tests Suite', () => {

  let comp: AppComponent;
  let fixture: ComponentFixture<AppComponent>;

  beforeEach(async(() => {
   TestBed.configureTestingModule({
      declarations: [ AppComponent ]
    })
    .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;

  });
});
```

So, as long as the component class is initiated for every test spec, it will have a new instance for every spec, and the internal scope will act based on that.

# Initializing the component

To test a component, it's important to initialize the component class so that we have the component object available to the test suite's scope and all the members of the object are available around the specific test suite.

As long as the component contains the template to render the UI, it's important to initialize the component before beginning e2e testing, and it depends on DOM elements.

So when we plan to do e2e testing for any component, we should initiate that in the DOM, as follows:

```
<body>
  <my-app></my-app>
</body>
```

# End-to-end versus unit test for components

In the preceding example, we looked at the component test suite, which is for unit testing, and we have to import and create an instance of the component class as the unit test. We will test the functionality or feature of every method that is defined in the component.

On the other hand, for end-to-end testing, we do not need to import or create an instance of the component class as we will not need to comment with the competent object or all its members. Rather, it needs to interact with the DOM elements of the landing page of the application that is running.

So, for that, we will need to run the application and navigate the test suite to the application landing page, and we can do so with the global `browser` object provided by Protractor itself.

Here is an example of what it should look like:

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

describe('Test suite for e2e test', () => {
    beforeEach(() => {
        browser.get('');
    });
});
```

We can navigate to all the URLs of the application using `browser.get('path')` as needed.

# Diving into our comment application

Now that the setup and approach have been decided, we can start our first test. From a testing point of view, as we will be using a top-down approach, we will write our Protractor tests first and then build the application. We will follow the same TDD life cycle we have already reviewed: test first, make it run, and make it better.

# Test first

The scenario given is already in a well-specified format and fits our Protractor testing template:

```
describe('', () => {
    describe('', () => {
     beforeEach(() => {
     });

      it('', () => {
      });
    });
});
```

Placing the scenario in the template, we get the following code:

```
describe('Given I am posting a new comment', () => {
    describe('When I push the submit button', () => {
        beforeEach(() => {
            // ...
        });

        it('Should then add the comment', () => {
            // ...
        });
    });
});
```

Following the 3 A's (assemble, act, assert), we will fit the user scenario in the template.

# Assemble

The browser will need to point to the first page of the application. As the base URL has already been defined, we can add the following to the test:

```
beforeEach(() => {
    browser.get('');
});
```

Now that the test has been prepared, we can move on to the next step: act.

# Act

The next thing we need to do, based on the user specification, is to add an actual comment. The easiest thing is to just put some text into an input box. The test for this, again without knowing what the element will be called or what it will do, is to write it based on what it should be.

Here is the code to add the comment section for the application:

```
beforeEach(() => {
    ...
    var commentInput = element(by.css('input'));
    commentInput.sendKeys('a sample comment');
});
```

The last assembly component, as a part of the test, is to click on the **Submit** button. This can be easily achieved in Protractor using the `click` function. Even though we don't have a page yet, or any attributes, we can still name the button that will be created:

```
beforeEach(() => {
    ...
    var submitButton = element(by.buttonText('Submit')).click();
});
```

Finally, we will hit the crux of the test and assert the user's expectations.

# Assert

The user expectation is that once the **Submit** button is clicked on, the comment is added. This is a little ambiguous, but we can determine that somehow the user needs to get notified that the comment was added.

The easiest approach is to display all comments on the page. In Angular, the easiest way to do this is to add an `*ngFor` object that displays all comments. To test this, we will add the following:

```
it('Should then add the comment', () => {
    var comment = element.all(by.css('li')).first();
    expect(comment.getText()).toBe('a sample comment');
});
```

Now the test has been constructed and meets the user specifications. It is small and concise. Here is the completed test:

```
describe('Given I am posting a new comment', () => {
    describe('When I push the submit button', () => {
      beforeEach(() => {
            //Assemble
            browser.get('');
            var commentInput = element(by.css('input'));
            commentInput.sendKeys('a sample comment');

            //Act
            var submitButton =  element(by.buttonText
            ('Submit')).click();
      });

       //Assert
  it('Should then add the comment', () => {
            var comment = element.all(by.css('li')).first();
            expect(comment.getText()).toBe('a sample comment');
  });
    });
});
```

# Make it run

Based on the errors and output of the test, we will build our application as we go.

Start the web server using the following command:

```
$ npm start
```

Run the Protractor test to see the first error:

```
$ protractor
```

Alternatively, we can run this:

```
$ npm run e2e // run via npm
```

Our first error could be that it's not getting the element the locator expected:

```
$ Error: Failed: No element found using locator:
By(css selector, input)
```

The reason for the error is simple: it's not getting the element as defined in the locator. We can see the current application and why it's not getting the element.

# Recap the present application

As long as we have cloned the sample Angular `quickstart` project as our application to test, it has a ready Angular environment. It Bootstraps the Angular project with a simple application component defined with `My First Angular 2 App` as the output.

So, in our TDD approach, we should not have any environment/Angular Bootstrap-related error, and it seems we are on the right path.

Let's take a look at what we have right now in our sample application. On our landing page, `index.html`, we have included all required library files and implemented `system.js` to load the application files.

In the `<body>` tag in the `index.html` file, we have initiated the application as follows:

```
<body>
    <my-app>Loading...</my-app>
</body>
```

The HTML tag expects a component with `my-app` as the selector for that component, and yes, we have that as `app.component.ts`, as follows:

```
import {Component} from '@angular/core';
@Component({
    selector: 'my-app',
    template: '<h1>My First Angular 2 App</h1>'
})
export class AppComponent { }
```

Angular introduced `ngModule` as an `appModule` to modularize and manage dependencies for every component. With this `appModule`, an application can define all the required dependencies at a glance. Besides that, it helped lazy-load the modules. We will look at the details of `ngModule` in the Angular docs.

It imports all the required modules in the application, declares all the modules from a single entry point, and also defines the Bootstrapping component.

The application always Bootstraps based on this file's configuration.

The file is located at the application root as `app.module.ts`, and it looks as follows:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent }  from './app.component';

@NgModule({
  imports:      [ BrowserModule],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

The entry point to the application is the `main.ts` file, which will import the `appModule` file and instruct to Bootstrap the application based on that:

```
import { platformBrowserDynamic } from '@angular/platform
-browser-dynamic';

import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

The test couldn't find our input locator. We need to add the input to the page, and we need to do that via the component's template.

# Adding the input

Here are the steps we need to follow to add the input to the page:

1. We will have to add a simple `input` tag to the template of the application component, as shown here:

```
template: `
<input type='text' />`
```

2. After running the test again, it seems that there are no more errors related to the input locator, but it has a new error as the `button` tag was missing:

```
$ Error: Failed: No element found using locator:
```

```
by.buttonText('Submit')
```

3. Just like the previous error, we need to add a `button` to the template with the appropriate text:

```
template: ` ..........
<button type='button'>Submit</button>`
```

4. After running the test again, it seems that there are no more errors related to the `button` locator, but again, it has a new error, as follows, as the repeater locator is missing:

```
$ Error: Failed: No element found using locator: By
(css selector, li)
```

This appears to be a result of our assumption that a submitted comment will be available on the page through `*ngFor`. To add this to the page, we will use a method in the component class to provide the data for the repeater.

# Component

As mentioned in the preceding section, the error occurs because there is no `comments` object. In order to add the `comments` object, we will use the component class that has an array of `comments` in its scope.

Perform the following steps to add a `comments` object to the scope:

1. As we already have `AppComponent` as a class in our component, we will need to define the comments array, which we can use in a repeater:

```
export class AppComponent {
    comments:Array<string>;
}
```

2. Then, we will add a repeater for the comments in the template, as shown here:

```
template: `.........
    <ul>
      <li *ngFor="let comment of comments">{{comment}}</li>
    </ul>`
```

3. Let's run the Protractor test and see where we are:

```
$   Error: Failed: No element found using locator: By(css
selector, li)
```

Oops! We are still getting the same error. However, don't worry; there might be some other issue.

Let's look at the actual page that gets rendered and see what's going on. In Chrome, navigate to `http://localhost:3000` and open the console to see the page source (*Ctrl + Shift + J*). Note that the repeater and component are both there; however, the repeater is commented out. Since Protractor is only looking at visible elements, it won't find the list.

Great! Now we know why the repeater list isn't visible, but we have to fix it. In order for a comment to show up, it has to exist on the component's `comments` scope.

The smallest change is to add something to the array to initialize it, as shown in the following code snippet:

```
export class AppComponent {
    comments:Array<string>;
    constructor() {
        this.comments = ['First comment', 'Second comment',
        'Third comment'];
    }
};
```

Now, if we run the test, we get the following output:

```
$ Expected 'First comment' to be 'a sample comment'.
```

Great, it seems we are getting closer as the errors have gone down! We have tackled almost all the unexpected errors and met our expectations.

So let's take a look at the changes that we have made so far and what our codes look like.

Here's the `body` tag of the `index.html` file:

```
<body>
    <my-app>Loading...</my-app>
</body>
```

The application component file is as follows:

```
import {Component} from '@angular/core';

@Component({
    selector: 'my-app',
    template: `<h1>My First Angular 2 App</h1>
    <input type='text' />
    <button type='button'>Submit</button>
    <ul>
```

```
        <li *ngFor="let comment of comments">{{comment}}</li>
    </ul>`
})
export class AppComponent {
    comments:Array<string>;

    constructor() {
        this.comments = ['First comment', 'Second comment',
        'Third comment'];
    }
}
```

# Make it pass

With TDD, we want to add the smallest possible component to make the test pass.

Since we have hard coded the comments array for the moment to be initialized to three items and the first items to be `First comment`, change `First comment` to `a sample comment`; this should make the test pass.

Here is the code to make the test pass:

```
export class AppComponent {
    comments:Array<string>;
    constructor() {
        this.comments = ['a sample comment', 'Second comment',
        'Third comment'];
    }
};
```

Run the test, and bam! We get a passing test:

```
$ 1 test, 1 assertion, 0 failures
```

Wait a second! We still have some work to do. Although we got the test to pass, it is not done. We added some hacks just to get it to pass. Two things stand out:

- We clicked on the **Submit** button, which really doesn't have any functionality
- We hardcoded the initialization of the expected value for a comment

The preceding changes are critical steps that we need to perform before we move forward. They will be tackled in the next phase of the TDD life cycle, that is, make it better (refactor).

# Make it better

The two components that need to be reworked are as follows:

- Adding behavior to the **Submit** button
- Removing the hardcoded value of the comments

# Implementing the Submit button

The **Submit** button needs to actually do something. We were able to sidestep the implementation by just hardcoding the value. Using our tried-and-trusted TDD techniques, switch to an approach focused on unit testing. So far, the focus has been on the UI and pushing changes to the code; we haven't written a single unit test.

For this next bit of work, we will switch gears and focus on driving the development of the **Submit** button through tests. We will be following the TDD life cycle (test first, make it run, and make it better).

## Configuring Karma

We did something very similar for the to-do list application in `Chapter 3`, *The Karma Way*. We won't spend as much time diving into the code, so review the previous chapters for a deeper discussion of some of the attributes.

Here are the steps we need to follow to configure Karma:

1. Update the `files` section with the added files:

```
files: [
    ...
    // Application files
    {pattern: 'app/**/*.js', included: false, watched:
    true}

    // Unit Test spec files
    {pattern: 'spec/unit/**/*.spec.js', included: false,
    watched: true}
    ...
],
```

2. Start Karma:

```
$ karma start
```

3. Confirm that Karma is running:

```
$ Chrome 50.0.2661 (Mac OS X 10.10.5): Executed 0 of 0
SUCCESS (0.003 secs / 0 secs)
```

## Test first

Let's start with a new file in the `spec/unit` folder, called `app.component.spec.ts`. This will contain the test spec for the unit test. We will use the base template, including all necessary imports, such as `TestBed`:

```
describe('', () => {
 beforeEach(() => {
 });

  it('', () => {
  });
});
```

According to the specification, when the **Submit** button is clicked on, it needs to add a comment. We will need to fill in the blanks of the three components of a test (assemble, act, and assert).

### Assemble

The behavior needs to be part of a component for the frontend to use it. The object under testing in this case is the component's scope. We need to add this to the assembly of this test. Like we did in `Chapter 3`, *The Karma Way*, we will do the same in the following code:

```
import {AppComponent} from "../../app/app.component";

describe('AppComponent Unit Test', () => {
    let comp: AppComponent;
    let fixture: ComponentFixture<AppComponent>;

    beforeEach(() => { fixture = TestBed.create
    Component(AppComponent);
      comp = fixture.componentInstance;

    });
});
```

Now, the component object and its member are available in the test suite and will be testable as expected.

#### Act

The specification determines that we need to call an `add` method in the component object. Add the following code to the `beforeEach` section of the test:

```
beforeEach(() => { comp.add('a sample comment');
});
```

Now, the assertion should get the first comment to test.

#### Assert

Assert that the comment items in the `component` object now contain any comment as the first element. Add the following code to the test:

```
it('',function(){
  expect(comp.comments[0]).toBe('a sample comment');
});
```

Save the file, and let's move on to the next step of the life cycle and make it run (execute).

## Make it run

Now that we have prepared the test, we need to make the test pass. Looking at the output of the console where Karma is running, we see the following:

```
$ TypeError: comp.add is not a function
```

Looking at our unit test, we see that this is the `add` function. Let's go ahead and put in an `add` function into the controller's `scope` object using the following steps:

1. Open the controller scope and create a function named `add`:

```
export class AppComponent {
    ............
    add() {
    // ....
    }
}
```

2. Check Karma's output, and let's see where we are:

```
$ Expected 'First comment' to be 'a sample comment'.
```

3. Now, we have hit the expectation. Remember to think of the smallest change to get this to work. Modify the `add` function to set the `$scope.comments` array to any comment when called:

```
export class AppComponent {
    ..............
    add() {
        this.comments.unshift('a sample comment');
    }
};
```

The `unshift` function is a standard JavaScript function that adds an item to the front of an array.

When we check Karma's output, we'll see the following:

```
$ Chrome 50.0.2661 (Mac OS X 10.10.5): Executed 1 of 1
SUCCESS (0.008 secs / 0.002 secs)
```

Success! The test passes, but again needs some work. Let's move on to the next stage and make it better (refactor).

## Make it better

The main point that needs to be refactored is the `add` function. It doesn't take any arguments! This should be straightforward to add, and simply confirms that the test still runs. Update the `add` function of `app.component.ts` to take an argument and use that argument to add to the `comments` array:

```
export class AppComponent {
    ..............
    add(comment) {
        this.comments.unshift(comment);
    }
};
```

Check the output window of Karma and ensure that the test still passes. The complete unit test looks as follows:

```
import {AppComponent} from "../../app/app.component";

describe('AppComponent Tests', () => {
    let comp: AppComponent;
    let fixture: ComponentFixture<AppComponent>;
```

```
    beforeEach(() => {
        fixture = TestBed.createComponent(AppComponent);
        comp = fixture.componentInstance;
        comp.add('a sample comment');
    });

    it('First item inthe item should match', () => {
        expect(comp.comments[0]).toBe('a sample comment');
    });
});
```

The `AppComponent` class file now looks like this:

```
import {Component} from '@angular/core';

@Component({
    selector: 'my-app',
    template: `<h1>My First Angular 2 App</h1>
    <input type='text' />
    <button type='button'>Submit</button>
    <ul>
      <li *ngFor="let comment of comments">{{comment}}</li>
    </ul>`
})
export class AppComponent {
    comments:Array<string>;

    constructor() {
        this.comments = ['First comment', 'Second comment',
        'Third comment'];
    }
    add(comment) {
        this.comments.unshift(comment);
    }
}
```

# Backing up the test chain

We completed the unit test and added the `add` function. Now we can add the function to specify the behavior of the **Submit** button. The way to link the `add` method to the button is to use the `(click)` event. The steps to add the behavior to the **Submit** button are as follows:

1. Open the `app.component.ts` file and update it as follows:

```
    @Component({
```

```
    template: `.......
     <button type="button" (click)="add('a sample
     comment')">Submit</button>
     ...........`
})
```

Hold on! Is the value hardcoded? Well, again, we want to make the smallest change and ensure that the test still passes. We will work through our refactors until the code is how we want it to be, but instead of a Big Bang approach, we want to make small, incremental changes.

2. Now, let's rerun the Protractor test and ensure that it still passes. The output says that it passed, and we are okay. The hardcoded value wasn't removed from the comments. Let's go ahead and remove that now.
3. The `AppComponent` class file should now look as follows:

```
constructor() {
    this.comments = [];
}
```

4. Run the test and see that we still get a passing test.

The last thing we need to mop up is the hardcoded value in `(click)`. The comment being added should be determined by the input in the comment input text.

# Binding the input

Here are the steps we need to follow to bind the input:

1. To be able to bind the input to something meaningful, add an `ngModel` attribute to the `input` tag:

```
@Component({
    template: `............
    <input type="text" [(ngModel)]="newComment">
    ...........`
})
```

2. Then, in the `(click)` attribute, simply use the `newComment` model as the input:

```
@Component({
   template: `.......
    <button type="button" (click)="add(newComment)">
    Submit</button>
```

```
        ..........`
    })
```

3. We will have to import the form module in the app module (`app.module.ts`) as it's a dependency for `ngModel`:

```
import { FormsModule }   from '@angular/forms';
@NgModule({
imports: [ BrowserModule, FormsModule ],
})
```

4. Run the Protractor test and confirm that everything has passed and is good to go.

# Onward and upward

Now that we have the first specification working, and it is end-to-end and unit tested, we can start the next specification. The next specification states that the user wants the ability to like a comment.

We will use the top-down approach and start our test from Protractor. We will continue to follow the TDD life cycle: test first, make it run, and make it better.

# Test first

Following the pattern, we will start with a basic Protractor test template:

```
describe('', () => {
     beforeEach(() => {
     });

      it('', () => {
      });
    });
```

When we fill in the specification, we get the following:

```
describe('When I like a comment', () => {
    beforeEach(() => {
    });

    it('should then be liked', () => {
      });
});
```

With the template in place, we are ready to construct the test.

# Assemble

The assembly of this test will require a comment to exist. Place the comment within the existing posted comment test. It should look similar to this:

```
describe(''Given I am posting a new comment', () => {
    describe('When I like a comment', () => {
    ...
    });
});
```

# Act

The user specification we test is that the **Like** button performs an action for a specific comment. Here are the steps that will be required and the code required to carry them out (note that the following steps will be added to the `beforeEach` text):

1. Store the first comment so that it can be used in the test:

```
var firstComment = null;
beforeEach(() => {
    ...
}
```

2. Find the first comment's `likeButton`:

```
var firstComment = element.all(by.css('li').first();
var likeButton = firstComment.element(by.buttonText('like'));
```

3. The code for the **Like** button when it is clicked on is as follows:

```
likeButton.click();
```

# Assert

The specification expectation is that once the comment has been liked, it is liked. This is best done by putting an indicator of the number of likes and ensuring that the count is `1`. The code will then be as follows:

```
it('Should increase the number of likes to one', () => {
var commentLikes = firstComment.element(by.binding('likes'));
```

```
    expect(commentLikes.getText()).toBe(1);
});
```

The created test now looks like this:

```
describe('When I like a comment', () => {
    var firstComment = null;
    beforeEach(() => {

      //Assemble
      firstComment = element.all(by.css('li').first();
      var likeButton = firstComment.element(by.buttonText('like'));

      //Act
      likeButton.click();
  });

  //Assert
  it('Should increase the number of likes to one', () => {
      var commentLikes = firstComment.element(by.css('#likes'));
      expect(commentLikes.getText()).toBe(1);
  });
});
```

# Make it run

The test has been prepared and is itching to run. We will now run it and fix the code until the test passes. The following steps will detail the error and fix the cycle required to make the test path:

1. Run Protractor.
2. View the error message in the command line:

```
$ Error: No element found using locator: by.buttonText("like")
```

3. As the error states, there is no **like** button. Go ahead and add the button:

```
@Component({
      template: `........
      <ul>
      <li *ngFor="let comment of comments">
      {{comment}}
    <button type="button">like</button>
      </li>
      </ul>`
  });
```

4. Run Protractor.
5. View the next error message:

```
$ Expected 'a sample comment like' to be 'a sample comment'.
```

6. By adding the **like** button, we caused our other test to fail. The reason is our use of the `getText()` method. Protractor's `getText()` method gets the inner text, including inner elements.
7. To fix this, we need to update the previous test to include **like** as part of the test:

```
it('Should then add the comment', () => {
  var comments = element.all(by.css('li')).first();
  expect(comments.getText()).toBe('a sample comment like');
});
```

8. Run Protractor.
9. View the next error message:

```
$ Error: No element found using locator: by.css("#likes")
```

10. It's time to add a `likes` binding. This one is a little more involved. The `likes` needs to be bound to a comment. We need to change the way the comments are held in the component. Comments need to hold the `comment` title and the number of likes. A comment should be an object like this:

```
{title:'',likes:0}
```

11. Again, the focus of this step is just to get the test to pass. The next step is to update the component's `add` function to create comments based on the object that we described in the preceding steps.
12. Open `app.component.ts` and edit the `add` function, as follows:

```
export class AppComponent {
    ......
      add(comment) {
          var commentObj = {title: comment, likes: 0};
          this.comments.unshift(commentObj);
      }
}
```

13. Update the page to use the value for the comment:

```
@Component({
    template: `...........
    <ul>
```

```
      <li *ngFor="let comment of comments">
  {{comment.title}}
    </li>
    </ul>`
})
```

14. Before rerunning the Protractor test, we need to add the new `comment.likes` binding to the HTML page:

```
@Component({
    template: `...........
    <ul>
      <li *ngFor="let comment of comments">
  {{comment.title}}
  ............
  <span id="likes">{{comment.likes}}</span>
      </li>
  </ul>`
})
```

15. Now rerun the Protractor tests, and let's see where the errors are:

```
$ Expected 'a sample comment like 0' to be 'a sample
comment like'
```

16. As the inner text of the comment has changed, we need to change the expectation of the test:

```
it('Should then add the comment',() => {
...
  expect(comments.getText()).toBe('a sample comment like 0');
});
```

17. Run Protractor:

```
$ Expected '0' to be '1'.
```

18. Finally, we are down to the expectation of the test. In order to make this test pass, the smallest change will be to make the **like** button update the likes on the `comment` array. The first step is to add a `like` method to the controller, which will update the number of likes:

```
export class AppComponent {
    ......
      like(comment) {
          comment.like++;
      }
```

```
}
```

19. Link the `like` method to the HTML page using a `(click)` attribute on the button, as follows:

```
@Component({
      template: `........
      <ul>
      <li *ngFor="let comment of comments">
      {{comment}}
    <button type="button" (click)="like(comment)">
    like</button>
<span id="likes">{{comment.likes}}</span>
      </li>
      </ul>`
  });
```

20. Run Protractor and confirm that the tests pass!

The page now looks like the following screenshot:


Compared with the diagram at the beginning of this chapter, all the features have been created. Now that we've made the test pass in Protractor, we need to check the unit tests to ensure that our changes didn't break the them.

# Fixing the unit tests

One of the primary changes required was to make the comment an object consisting of a value and the number of likes. Before thinking too much about how the unit tests could have been affected, let's kick them off. Execute the following command:

```
$ karma start
```

As expected, the error is related to the new `comment` object:

```
$ Expected { value : 'a sample comment', likes : 0 } to be
'a sample comment'.
```

Reviewing the expectation, it seems like the only thing required is for `comment.value` to be used in the expectation, as opposed to the `comment` object itself. Change the expectation as follows:

```
it('',() => {
    var firstComment = app.comments[0];
    expect(firstComment.title).toBe('a sample comment');
})
```

Save the file and check the Karma output. Confirm that the test passes. Both the Karma and Protractor tests pass, and we have completed the primary user behaviors of adding a comment and liking it. We are now free to move on to the next step and make things better.

# Make it better

All in all, the approach ended with the result we wanted. Users are now able to like a comment in the UI and see the number of likes. The major thing from a refactor standpoint is that we have not unit-tested the `like` method.

Reviewing our development to-do list, we see that the list is an action that we wrote down. Before completely wrapping up the feature, let's discuss the option of adding a unit test for the `like` functionality.

# Coupling the test

As already discussed, tests are tightly coupled to the implementation. This is a good thing when there is a complicated logic involved or we need to ensure that certain aspects of the application behave in certain ways. It is important to be aware of the coupling and know when it is important to bring it into the application and when it is not. The `like` function, which we created, simply increments a counter on an object. This can be easily tested; however, the coupling that we will bring in with a unit test will not give us the extra value.

In this case, we will not add another unit test for the `like` method. As the application progresses, we may find the need to add a unit test in order to develop and extend the function.

Here are some things I consider when adding a test:

- Does adding a test outweigh the cost of maintaining it?
- Is the test adding value to the code?
- Does it help other developers understand the code better?
- Is the functionality being tested in some or the other way?

Based on our decision, there is no more refactoring or testing required. In the next section, we will take a step back and review the main points of this chapter.

# Self-test questions

Q1. Karma needs the Selenium WebDriver to run tests.

- True
- False

Q2. Given the following code segment, how would you select the following button:

```
<button type="button">Click Me</button>?
```

- `element.all(by.button('button'))`
- `element.all(by.css('type=button'))`
- `element(by.buttonText('Click Me')`

# Summary

In this chapter, we walked through the TDD techniques of using Protractor and Karma together. As the application was developed, we were able to see where, why, and how to apply TDD testing tools and techniques.

The approach, top-down, was different from the bottom-up approach discussed in `Chapter 3`, *The Karma Way*, and `Chapter 4`, *End-to-End Testing with Protractor*. With the bottom-up approach, the specifications are used to build unit tests and then build the UI layer on top of that. In this chapter, a top-down approach was shown to focus on the user's behavior.

The top-down approach tests the UI and then filters the development through the other layers. Both the approaches have their merits. When applying TDD, it is essential to know how to use both. In addition to walking through a different TDD approach, we saw some of the core testing components of Angular, such as the following:

- Testing a component from end-to-end and unit perspectives
- Importing a component class to the test suite and initiating it for unit testing
- Protractor's ability to bind to `ngModel`, send keystrokes to input columns, and get an element's text by its inner HTML code and all subelements

The next chapter will build on the techniques used here and look into headless browser testing, advanced techniques for Protractor, and how to test Angular routes.

# 7
# Flip Flop

At this point, we should be confident about performing the initial implementation of an Angular application using TDD. Also, we should be familiar with using the test-first approach. The test-first approach is very good for the learning stage, but sometimes it's a time suck when we get a lot of errors. For simple and known behavior, it may not be good to go for the test-first approach.

We have already seen how the test-first approach works, so we can skip those steps by checking any feature without creating those components. Besides that, we can go one step further to make us more confident in writing our components faster. We can have our components ready and then write end-to-end test specs to test the expected behavior. If the e2e test fails, we can trigger an error in the Protractor debugger.

In this chapter, we will continue to expand our knowledge of applying TDD (but not the test-first approach) with Angular. We will not discuss the details of the basic Angular component ecosystem here; rather, we will go a step ahead and include more Angular features. We will take our knowledge further by looking at the following topics:

- Angular routes
- Navigating to routes
- Communicating with route parameter data
- Protractor location references with CSS and HTML elements

# Fundamentals of TDD

In this chapter, we will walk-through applying TDD to routes and navigation for a search application. Before getting into the walk-through, we need to be aware of some of the techniques, configurations, and functions that will be used throughout this chapter, which include the following:

- Protractor locators
- Headless browser testing

After reviewing these concepts, we can move on to the walk-through.

# Protractor locators

Protractor locators are key components where everybody should spend some time to learn. In the previous Protractor chapters, we understood some commonly used locators with working examples. We will provide some examples of the Protractor `Locator` here.

Protractor locators allow us to find elements within an HTML page. In this chapter, we will see the CSS, HTML, and Angular-specific locators in action. Locators are passed to the `element` function. The `element` function will find and return the elements on a page. The generic locator syntax is as follows:

```
element(by.<LOCATOR>);
```

In the preceding code, `<LOCATOR>` is a placeholder. The following sections describe a couple of these locators.

## CSS locators

CSS is used to add layout, color, formatting, and style to an HTML page. From an end-to-end testing perspective, the look and style of an element may be part of a specification. As an example, consider the following HTML snippet:

```
<div class="anyClass" id="anyId"></div>
// ...
var e1 = element(by.css('.anyClass'));
var e2 = element(by.css('#anyId'));
var e3 = element(by.css('div'));
var e4 = $('div');
```

All these four selections will select the `div` element.

## Button and link locators

Besides being able to select and interpret the way something looks, it is also important to be able to find buttons and links within a page. This will allow a test to interact with the site easily. Here are a couple of examples:

- `buttonText` locator:

```
<button>anyButton</button>
// ...
var b1 = element(by.buttonText('anyButton'));
```

- `linkText` locator:

```
<a href="#">anyLink</a>
// ...
var a1 = element(by.linkText('anyLink'));
```

## URL location references

When testing Angular routes, we need to be able to test the URL of our test. By adding tests around the URL and location, we have to ensure that the application works with specific routes. This is important because routes provide an interface to our application. Here is how to get the URL reference in a Protractor test:

```
var location = browser.getLocationAbsUrl();
```

Now that we have seen how to use the different locators, it is time to put the knowledge to use.

# Preparing an Angular project

It is important to get a process and method to set up your projects quickly. The less time you spend on thinking about the structure of the directory and the required tools, the more time you can spend developing!

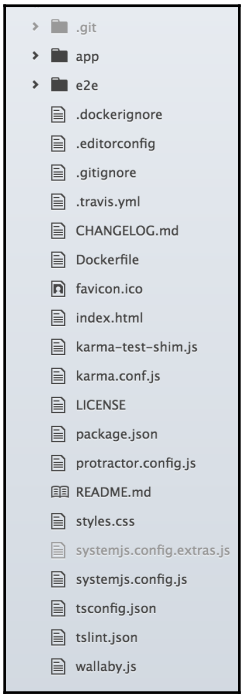
For this reason, in the previous chapters, we looked at how to get simple existing projects for Angular developed as `quickstart` projects (`https://github.com/angular/quickstart`).

However, some people use the `angular2-seed` (`https://github.com/mgechev/angular2-seed`) project, Yeoman, or create a custom template. Although these techniques are useful and have their merits, when starting out in Angular, it is essential to understand what it takes to build an application from the ground up. By building the directory structure and installing tools by ourselves, we will understand Angular better.

You will be able to make layout decisions based on your specific application and needs, as opposed to fitting them into some other module. As you grow and become a better Angular developer, this step may not be needed and will become second nature to you.

# Loading the existing project

To start off, we will clone the project from the Angular `quickstart` project at `https://github.com/angular/quickstart`, rename it as `angular-flip-flop`, and our project folder structure will look as follows:




In the previous chapters, we discussed how to set up the project, understood the different components involved, and walked through the entire process. We will skip these details and assume that you can recall how to perform the necessary installation.

# Preparing the project

This `quickstart` project doesn't included the base `href` in the project's landing page (`index.html`). We will need that to proceed perfectly with routing, so let's add a single line (`base href`) to the `<head>` section of `index.html`:

```
<base href="/">
```

Here, our bootstrapping component is in the application component and the HTML template is in the component itself. We should separate the template to a new file before proceeding.

For that, we will update our application component (`app/app.component.ts`), as follows:

```
import { Component } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'my-app',
  templateUrl: 'app.component.html'
})
export class AppComponent {

};
```

Let's create our separate template file at `app/app.component.html`. And the code will look like this:

```
<h1>My First Angular 2 App</h1>
```

# Running the project

Let's proceed with it and get ready to run using the following commands:

```
$ cd angular-flip-flop
$ npm install // To install the required node modules.
$ npm run // To build and run the project in http server.
```

To confirm the installation and run the project, the application will automatically run in the web browser.

Here is the expected output after running the project:

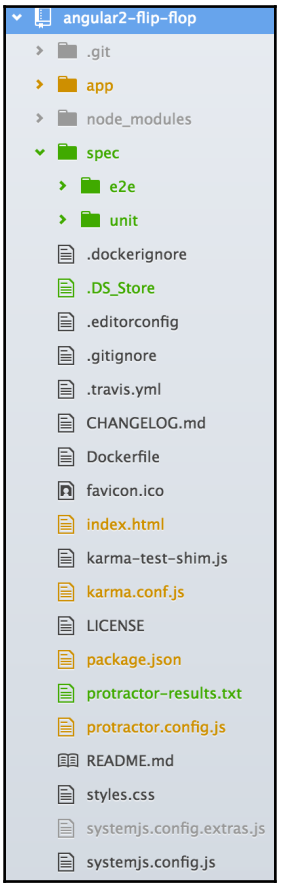

# Restructuring the project

Let's change the project structure a bit, not much though. By default, it has included unit testing in the same location as the components file and separated the e2e testing file outside the `app/` folder in the `e2e/` folder.

However, we will keep all the tests in the same location, that is, outside of `app`; we will keep all the tests as `spec/e2e` and `spec/unit`.

The goal is to separate the test specs from the component. This way, we can keep our unit tests file outside in the `spec/unit` folder.

So, our current folder structure will look like this:




As long as we have changed the path for both unit and e2e tests, we should have to change the path in the Karma configuration and Protractor configuration files.

# Setting up headless browser testing for Karma

In the previous chapters, we were running Karma using the default configuration. The default Chrome configuration launches Chrome on every test. Testing against the actual code and browser that the application will run in is a powerful tool. However, when launching, a browser may not always know how you want it to behave. From a unit test perspective, you may not want the browser to be launched in a window. The reason could be that tests may take a long time to run or you may not always have a browser installed.

Luckily, Karma comes equipped with the ability to easily configure PhantomJS, a headless browser. A headless browser runs in the background and will not display web pages in a UI. The PhantomJS headless browser is a really great tool to use for testing. It can even be set up to take screenshots of your tests! Read more about how this is done and about the WebKit used on the PhantomJS site at `http://phantomjs.org/`. The following setup configuration will show us how to set up PhantomJS with Karma for headless browser testing.

## Preconfiguration

When Karma is installed, it automatically includes the PhantomJS browser plugin. For further reference, the plugin is located at `https://github.com/karma-runner/karma-phantomjs-launcher`. There shouldn't be any more installation or configuration required.

However, if your setup states that it is missing `karma-phantomjs-launcher`, you can easily install it using `npm`, like this:

```
$ npm install karma-phantomjs-launcher --save -dev
```

## Configuration

PhantomJS is configured in the `browsers` section of the Karma configuration. Open the `karma.conf.js` file and update it with the following details:

```
browsers: ['PhantomJS'],
```

Do that in the `plugins` option as well:

```
plugins: [
```

```
        .........
        require('karma-phantomjs-launcher'),
    ],
```

Now that the project has been initialized and configured with headless browser testing, you can see it in action through the following walk-throughs.

# Walk-through of Angular routes and navigation

This walk-through will leverage Angular routes. Routes are an extremely useful feature of Angular, as they were in Angular 1.x before it, but more powerful. They allow us to control certain aspects of the application using different components.

This walk-through will flip between components to show us how to use TDD to build routes. The following are the specifications. There will be a navigation menu, which has two menu items, **View1** and **View2**:

- In the navigation menu, click on label **View1**
- The content area (router outlet) will load/flip **View1** content

And here's the second part:

- In the navigation menu, click on label **View2**
- The content area (router outlet) will load/flip **View2** content

Essentially, this will be an application that does a flip/flop between two views.

# Setting up Angular routes

The router is an optional service in Angular, so it's not included in the Angular core. If we need to use the router, we will have to install the Angular `router` service in our application.

As long as we have cloned our project from `quickstart`, we should be okay because it has recently added the Angular router to its dependencies, but we should check and confirm. If it doesn't have `@angular/router` in its dependencies in `package.json`, we can install the Angular router using `npm`, as follows:

```
$ npm install @angular/router --save
```

# Defining directions

A route specifies a location and expects a result. From an Angular perspective, the routes must first be specified and then associated with certain components.

To implement a router in our application, we will need to import the router module in the application module, where it will register the router in the application. After that, we will need to configure all the routes and pass that configuration to the application module.

# The router module

To implement a router in the application, we will need to import the `RouterModule` in our application module, located at `app/app.module.ts`, as follows:

```
import {RouterModule} from "@angular/router";
```

This will just make the `router` module available in the application system, but we will have to have a router configuration to define all the possible routers in the entire application and then import that configuration to the application ecosystem via the application module.

# Configuring routes

A router is useless until we configure it, and to do so, we first need to import the `router` component. Configuration will mainly contain a list of arrays, where route paths and related components live as key-value pairs. We can add the configuration array to the application module, or we can create a separate configuration file and include the app module in it. We will go for the second option so that route configuration will be separated from the app module.

Let's create the router configuration file in the application root as `app/app.routes.ts`. There, at first, we will need to import the Angular `Routes` from the Angular service, as shown here:

```
import {Routes} from '@angular/router';
```

From the router config file, we will need to export the configuration array, as follows:

```
export const rootRouterConfig: Routes = [
 // List of routes will come here
];
```

# Routers in the application

We've already imported the `router` module to our application module, located at `app/app.module.ts`.

Then, we will need to import the router configuration file (`rootRouterConfig`) to this application module file, as follows:

```
import {rootRouterConfig} from "./app.routes";
```

In the application module, we know `NgModule` imports the optional modules to the application ecosystem, and similarly, to include the router in the application, `RouterModule` has a function known as `RouterModule.forRoot(RouterConfig)`, which accepts the `routerConfiguration` to implement a router in the entire application.

The application module (`app/app.module.ts`) will import that `RouterModule` as follows:

```
@NgModule({
  declarations: [AppComponent, ........],
  imports     : [........., RouterModule.forRoot(rootRouterConfig)],
  bootstrap   : [AppComponent]
})
export class AppModule {
}
```

# Routes in the config

Now, let's add some routes to our `Routes` configuration array, which is located at `app/app.routes.ts`. The route configuration array contains some objects as key-value pairs, with mostly two to three elements in every object.

The first element in the array object contains the `path`, and the second one contains the relevant `component` to load for that `path`.

Let's add two routes to our configuration array, as shown here:

```
export const rootRouterConfig: Routes = [
  {
    path: 'view1',
    component: View1Component
  },
  {
    path: 'view2',
    component: View2Component
  }
```

```
];
```

Here, two routes, `view1` and `view2`, are defined, and two components have been assigned to load for that route.

In some cases, we may need to redirect from one route to another. For example, for the root path of the application (`''`), we may plan to redirect to the `view1` route. For that, we have to set the `redirectTo` element in the object and assign some route name as its value. We will also need to add one extra element as `pathMatch` and set its value to `full` so it will match the full path before redirecting to some other route.

The code will look as follows:

```
export const rootRouterConfig: Routes = [
  {
    path: '',
    redirectTo: 'view1',
    pathMatch: 'full'
  },
  ..............
];
```

So, yes, our initial route configuration is ready to go. Now, the full configuration will look like this:

```
import {Routes} from '@angular/router';
import {View1Component} from './view/view1.component';
import {View2Component} from './view/view2.component';

export const rootRouterConfig: Routes = [
  {
    path: '',
    redirectTo: 'view1',
    pathMatch: 'full'
  },
  {
    path: 'view1',
    component: View1Component
  },
  {
    path: 'view2',
    component: View2Component
  }
];
```

I should mention here that we have to import the `view1` and `view2` components as we have used them in the router config.

To learn more in detail about Angular routes, refer to `https://angular.io/docs/ts/latest/guide/router.html`.

# Hands-on routes

So far, we have installed and imported a router module, configured routes, and included things in the application ecosystem. We still need to do some related tasks, such as creating a router outlet, creating navigation, and creating the component defined in the route, to have hands-on experience with routes.

## Defining the router outlet

As long as the route is configured in `appComponent`, we need a placeholder to load the route's navigated components, which Angular defines as the route outlet.

A `RouterOutlet` is a placeholder that Angular dynamically fills based on the application's route.

For our application, we will place the `router-outlet` in the `appComponent` template, located at (`/app/app.component.html`), like this:

```
<router-outlet></router-outlet>
```

## Preparing the navigation

In the route configuration, we have set two paths, `/view1` and `/view2`, for our application. Now, let's create the navigation menu with two route paths to make navigation easy. For that, we can create a separate simple component so that navigation can be isolated for the entire application component.

Create a new component file for the `NavbarComponent` at `/app/nav/navbar.component.ts`, as shown here:

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-navbar',
  templateUrl: 'navbar.component.html',
```

```
  styleUrls: ['navbar.component.css']
})
export class NavbarComponent {}
```

Also, create a template for the navigation component at (`/app/nav/navbar.component.html`), as shown here:

```
<main>
  <nav>
    <a [routerLink]="['/view1']">View1</a>
    <a [routerLink]="['/view2']">View2</a>
    <a [routerLink]="['/members']">Members</a>
  </nav>
</main>
```

Don't bother about the `members` link in in the navigation for now; I'll tell you what it is in a later section.

Let's create the basic CSS style for the navigation component for a better look at `/app/nav/navbar.component.css`, as shown here:

```
:host {
  border-color: #e1e1e1;
  border-style: solid;
  border-width: 0 0 1px;
  display: block;
  height: 48px;
  padding: 0 16px;
}

nav a {
  color: #8f8f8f;
  font-size: 14px;
  font-weight: 500;
  margin-right: 20px;
  text-decoration: none;
  vertical-align: middle;
}

nav a.router-link-active {
  color: #106cc8;
}
```

We have a navigation component. Now we will have to bind that to our app component, which is our application landing page.

To do so, we have to append the following to the `appComponent` template, located at `/app/app.component.html`:

```
<h1>My First Angular 2 App</h1>
<app-navbar></app-navbar>
<router-outlet></router-outlet>
```

# Preparing the components

For each defined route, we will need to create an individual component as every route will be associated with a component.

Here, we have two defined routes, and we will need to create two individual components to work on the routes' navigation. We will create `View1Component` and `View2Component` as per our requirement.

Create a new component file for the `View 1` component at `/app/view/view1.component.ts`, as follows:

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-view1',
  template: '<div id="view1">I am view one component</div>'
})
export class View1Component { }
```

Create another component file for the `View 2` component (`/app/view/view2.component.ts`):

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-view2',
  template: '<div id="view2">I am view two component</div>'
})
export class View2Component { }
```

We are ready with our routes and related components (`Navigation`, `View1`, and `View2`). Hopefully, everything's working as expected and we can see the application output in the browser.

Wait, before looking at the expected output in the browser, let's test the expected result with end-to-end testing. Now we know the expected behavior, and we will write out the e2e test specs based on our expectation. Once we're ready with the e2e test specs, we will see how it fulfills our expectation.

# Assembling the flip/flop test

Following the first of the 3 As, *assemble*, these steps will show us how to assemble the test:

1. Start with the Protractor base template, as follows:

```
describe('Given views should flip through navigation
interaction', () => {
  beforeEach( () => {
    // .....
});

it('Should fliped to the next view', () => {
   // ....
});
});
```

2. Navigate to the root of the application using the following code:

```
browser.get('view1');
```

3. The `beforeEach` method needs to confirm that the correct component's view is being displayed. This can be done using a CSS locator to look for the `div` tag of `view1`. The expectation will look as follows:

```
var view1 = element(by.css('#view1'));
expect(view1.isPresent()).toBeTruthy();
```

4. Then, add an expectation that `view2` is not visible:

```
var view2 = element(by.css('#view2'));
expect(view2.isPresent()).toBeFalsy();
```

5. Then add further confirmation by getting the entire text of the `view1` component:

```
var view1 = element(by.css('#view1'));
expect(view1.getText()).toEqual('I am view one component');
```

# Flipping to the next view

The preceding test needs to confirm that when the `view2` link will be clicked in the navigation, the `view2` component's content will load. In order to test this, we can use the `by.linkText` locator. Here is what it will look like:

```
var view2Link = element(by.linkText('View2'));
view2Link.click();
```

The `beforeEach` function is now complete and looks like this:

```
var view1 = element(by.css('#view1'));
var view2 = element(by.css('#view2'));
beforeEach(() => {
    browser.get('view1');
    expect(view1.isPresent()).toBeTruthy();
    var view2Link = element(by.linkText('View2'));
    view2Link.click();
})
```

Next, we will add the assertion.

# Asserting a flip

The assertion will again use Protractor's CSS locator, as shown here, to find whether `view2` is available:

```
it('Should fliped to View2 and view2 should visible', () => {
  expect(view2.isPresent()).toBeTruthy();
});
```

We also need to confirm that `view1` is no longer available. Add the expectation that `view1` should not exist, as follows:

```
it('Should fliped to View2 and view1 should not visible', () => {
  expect(view1.isPresent()).toBeFalsy();
});
```

Also, to make sure, we can check whether the `view2` contents have been loaded, as shown here:

```
it('Should fliped to View2 and should have body content as expected',  ()
=> {
    expect(view2.getText()).toEqual('I am view two component');
});
```

As we have the test about to switch from the `view1` to `view2` component by clicking on the `view2` link in the navigation, let's go back to the `view1` component by clicking on the `view1` link in the navigation, hoping things work as expected:

```
it('Should flipped to View1 again and should visible', () => {
    var view1Link = element(by.linkText('View1'));
    view1Link.click();
    expect(view1.isPresent()).toBeTruthy();
    expect(view2.isPresent()).toBeFalsy();
  });
```

The test has now been assembled.

# Running the flip/flop test

Our test spec is ready, and it's time to run it and see the result.

First, we will have to keep our project running via the HTTP server, with the following command:

```
$ npm start
```

Then, we have to run Protractor. Be sure about the port number of the running application and Protractor configuration file; just to be sure, update the running server port in the configuration. To run Protractor, use the following command:

```
$ npm run e2e
```

The results should be as follows:

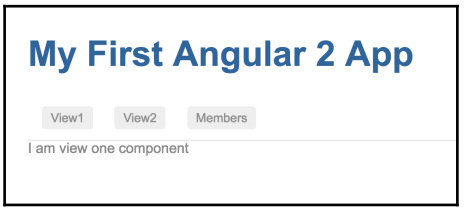
```
Suite: Given views should flip through navigation in
    passed - View1 should have body content as expected
    passed - Should flipped to View2 and view2 should visible
    passed - Should flipped to View2 and should have body content
    as expected
    passed - Should flipped to View1 again and should visible
        Suite passed: Given views should flip through navigation in
```

The Protractor tests have passed as per our expectations. Now we can have a look at the browser to check whether things are working the same way as the e2e test results.

# Opening the app in a browser

As long as we've run the `npm start` command for e2e testing, our application could run on a specific port, `3000`, on the localhost. By default, it will open in the browser.

The expected output is shown in the following screenshot:




# Searching the TDD way

This walk-through will show us how to build a simple search application. It has two components: the first discusses a search query component, and the second uses routes to display search result details.

# Walk-through of the search query

The application being built is a search application. The first step is to set up the search area with search results. Imagine that I am performing a search. In this case, the following actions will occur:

- A search query is typed in
- The results are displayed at the bottom of the search box

This piece of the application is very similar to the test, layout, and approach we have seen in `Chapter 6`, *The First Step*. The application will need to use an input, respond to a click, and confirm the resulting data. Since the tests and code use the same functionality as the previous example, it is not worth providing a complete walk-through of the search functionality. Instead, the following subsections will show the required code with a few explanations.

# The search query test

The following code represents the test for the search query functionality:

```
describe('Given should test the search feature', () => {
    let searchBox, searchButton, searchResult;

    beforeEach(() => {
    //ASSEMBLE
    browser.get('');
    element(by.linkText('Search')).click();
    searchResult = element.all(by.css('#searchList tbody tr'));
    expect(searchResult.count()).toBe(3);

    //ACT
    searchButton = element(by.css('form button'));
    searchBox = element(by.css('form input'));
    searchBox.sendKeys('Thomas');
    searchButton.click();
    });

    //Assert
    it('There should be one item in search result', () => {
    searchResult = element.all(by.css('#searchList tbody tr'));
    expect(searchResult.count()).toBe(1);
    });
});
```

We should notice a parallel to the previous tests. The functionality is written to mirror the behavior of a user typing in the search box. The test finds the input field, types a value, and then selects the button that says **Search**. The assertion confirms that the result contains a single value.

# The search application

To perform a search operation, we will need to create a search component that will contain an input field to accept the user input (search query) and a button to perform user action with a click event. Besides that, it may have a placeholder to contain the search result.

As long as our application already has the router included, we can place the search component for a specific route.

Note that we have called our search component as `MembersComponent`, since we worked with some member data in the search component. And routes will be configured based on that as well.

So, in our existing `app.routes.ts` file, we will add the following search routes:

```
export const rootRouterConfig: Routes = [
  {
    path: '/members',
    component: MembersComponent
  }
.....................
];
```

# The search component

The search component (`MembersComponent`) will be the main class for the search functionality here. It will perform a search and return the search result.

During the initial loading of the search component, it will not have any search query, so we have set the behavior to return all the data. Then, after the search trigger, it will return data based on a search query.

The search component will be placed in `app/members/members.compoennt.ts`. In the code, at first, we will have to import the required Angular services, as shown here:

```
import { Component, OnInit } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Router } from '@angular/router';
```

We will use the `Http` service for the AJAX call, and by default, in Angular, the `Http` service returns an observable object. However, it's easier to handle a promise than an observable object. So, we will convert this observable object to a promise. Angular suggests using the `rxjs` module, which has the `toPromise` method, used to convert an observable object to a promise. So, we will import the `rxjs` module, as follows:

```
import 'rxjs/add/operator/toPromise';
```

Angular introduced the `ngOnInit()` method to be executed while initializing the component, similar to the contractor method in any class, but it's helpful to run the test spec. For that, we have imported the `OnInit` interface from the Angular core, and the `Component` class will implement the `OnInit` interface to get the `ngOnInit` method.

Besides that, the `Component` class should inject the required module, such as `Http` and `Router`, as follows:

```
export class MembersComponent implements OnInit {
    constructor(private http:Http, private router:Router) {
```

```
    }
}
```

As discussed, we will use the `ngOnInit()` method, and from that, we will initialize the searching mechanism, like this:

```
export class MembersComponent implements OnInit {
 ngOnInit() {
    this.search();
  }
```

Here, we will apply the `search` feature on a member list, and for that, we have some dummy data at `app/data/people.json`. We will retrieve the data from here and perform a search operation on the data. Let's see how:

- The `getData()` method will retrieve the data from the API and will return a promise.

```
getData() {
    return this.http.get('app/data/people.json')
    .toPromise()
    .then(response => response.json());
}
```

- The `searchQuery()` method will resolve the returned promise and will make a data array based on the search query. If no search query is provided, it will return the complete dataset as an array:

```
searchQuery(q:string) {
    if (!q || q === '*') {
      q = '';
    } else {
      q = q.toLowerCase();
    }
    return this.getData()
      .then(data => {
      let results:Array<Person> = [];
      data.map(item => {
        if (JSON.stringify(item).toLowerCase().includes(q)) {
          results.push(item);
        }
      });
      return results;
    });
}
```

- The `search()` method will prepare the dataset for the template to bind to in the frontend:

```
search(): void {
  this.searchQuery(this.query)
  .then(results => this.memberList = results);
}
```

We have one more optional method here, which is used to navigate to the member details component. We've called that the `person` component. Here, the `viewDetails()` method will pass the member ID, and the `router.navigate()` method will navigate the application to the `person` component with the ID as a parameter, as shown here:

```
viewDetails(id:number) {
    this.router.navigate(['/person', id]);
  }
```

The complete code of `MembersComponent` will be as follows:

```
import { Component, OnInit } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Router } from '@angular/router';
import 'rxjs/add/operator/toPromise';
import { Person } from './person/person.component';

@Component({
  selector: 'app-member',
  moduleId: module.id,
  templateUrl: 'members.component.html',
  styleUrls: ['members.component.css']
})
export class MembersComponent implements OnInit {
  memberList: Array<Person> = [];
  query: string;

  constructor(private http:Http, private router:Router) {
  }

  ngOnInit() {
    this.search();
  }

  viewDetails(id:number) {
    this.router.navigate(['/person', id]);
  }

  getData() {
```

```
    return this.http.get('app/data/people.json')
      .toPromise()
      .then(response => response.json());
  }

  search(): void {
    this.searchQuery(this.query)
    .then(results => this.memberList = results);
  }

  searchQuery(q:string) {
    if (!q || q === '*') {
      q = '';
    } else {
      q = q.toLowerCase();
    }
    return this.getData()
      .then(data => {
      let results:Array<Person> = [];
      data.map(item => {
        if (JSON.stringify(item).toLowerCase().includes(q)) {
          results.push(item);
        }
      });
      return results;
    });
  }
}
```

The `search` component template contains the search form and the list of search results when there are results to show.

The template looks as follows:

```
<h2>Members</h2>

<form>
  <input type="search" [(ngModel)]="query" name="query"
(keyup.enter)="search()">
  <button type="button" (click)="search()">Search</button>
</form>

<table *ngIf="memberList" id="searchList">
  <thead>
  <tr>
    <th>Name</th>
    <th>Phone</th>
  </tr>
```

```
    </thead>
    <tbody>
    <tr *ngFor="let member of memberList; let i=index">
      <td><a href="javascript:void(0)"
(click)="viewDetails(member.id)">{{member.name}}</a></td>
      <td>{{member.phone}}</td>
    </tr>
    </tbody>
  </table>
```

The preceding Angular components are similar to what has already been shown in the previous chapters.

We are using a dummy dataset from the `people.json` file, which contains information about people with addresses. We want to split the information in two parts, one as summary information and the other with address details. As we will work with this dataset, it will be easy to make an object model for this dataset.

The summary dataset will be defined as a `Person` object, and address details will be defined as `Address`. Let's create a person object at `app/members/person/person.component.ts` and place both object models in the same file.

The two object model classes for `Person` and `Address` look like this:

```
export class Person {
  id:number;
  name:string;
  phone:string;
  address:Address;

  constructor(obj?:any) {
    this.id = obj && Number(obj.id) || null;
    this.name = obj && obj.name || null;
    this.phone = obj && obj.phone || null;
    this.address = obj && obj.address || null;
  }
}

export class Address {
  street:string;
  city:string;
  state:string;
  zip:string;
```

```
    constructor(obj?:any) {
      this.street = obj && obj.street || null;
      this.city = obj && obj.city || null;
      this.state = obj && obj.state || null;
      this.zip = obj && obj.zip || null;
    }
  }
```

# Show me the search results!

Now that the **Search** button is set with the required features, the result should contain the data only based on the search query, instead of everything. Let's look at the user specification.

Given a set of search results:

- We will have the member list based on the search query
- We will click on any member's name and navigate to the details component for details

Following the top-down approach, the first step will be the Protractor test, followed by the necessary steps to get the application fully functional.

## Testing the search results

As the specification states, we will need to leverage the existing search results. Instead of creating a test from scratch, we can add to the existing search query test. Start with a base test embedded in the search query test, as follows:

```
describe('Given should test the search result in details view', () => {
  beforeEach(() => {
  });

  it('should be load the person details page', () => {
  });
});
```

The next step is building the test.

## Assembling the search result test

In this case, the search results are already available from the search query test. We don't have to add any more setup steps for the test.

## Selecting a search result

The object under testing is the result. The test is that the result is selected and the application must then do something. The steps to writing this in Protractor are as follows:

1. Select the `resultItem`. As we will be representing the details using a route, we will create a link to the details page and click on the link. Here is how to create a link:

   Select the link within the `resultItem`. This uses the currently selected element and then finds any subelements that meet the criteria. The code for this is as follows:

   ```
   let resultItem = element(by.linkText('Demaryius Thomas'));
   ```

2. Now, to select the link, add the following code:

   ```
   resultItem.click();
   ```

## Confirming a search result

Now that the search item has been selected, we will need to verify that the result details page is visible. The simplest solution at this point is to ensure that the details view is visible. This can be done using Protractor's CSS locator to look for the search detail view. The following is the code to be added for confirming a search result:

```
it('Should be load the person details page', () => {
    var resultDetail = element(by.css('#personDetails'))
    expect(resultDetail.isDisplayed()).toBeTruthy();
})
```

Here is the complete test:

```
describe('Given should test the search result in details view', () => {

  beforeEach(() => {
    browser.get('members');
    let searchButton = element(by.css('form button'));
    let searchBox = element(by.css('form input'));
```

```
    searchBox.sendKeys('Thomas');
    searchButton.click();
    let resultItem = element(by.linkText('Demaryius Thomas'));
    resultItem.click();
  });

  it('should be load the person details page', () => {
    var resultDetail = element(by.css('#personDetails'))
    expect(resultDetail.isDisplayed()).toBeTruthy();
  });

});
```

Now that the test is set up, we can continue to the next phase of the life cycle and run it.

# The search result component

The search result component (the one we named `Person`) will route to accept the person ID from the `params` route and will search data based on that ID.

The search result component will be placed in `app/members/person/person.component.ts`. In the code, at first, we will have to import the required Angular services, as shown here:

```
import { Component, OnInit } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Router, ActivatedRoute, Params } from '@angular/router';
```

We've already seen some of these Angular services in the `members` component. Here, we will mainly discuss the `ActivatedRoute`, as it's new. This is an Angular router module, which is used to interact with the current/activated route: when we need to access the `params` in the current route, we'll access them through this.

As we discussed, we will need `ActivatedRoute` while initializing the component; therefore, we have called `ActivatedRoute` on the `ngOnInit()` method. It will provide the current route params for us, and we will have our expected ID, which will be used to retrieve that specific `Person` from the demo members dataset, as shown here:

```
export class PersonComponent implements OnInit {
  person: Person;
  constructor(private http:Http, private route: ActivatedRoute,
  private router: Router) {
  }

  ngOnInit() {
```

```
      this.route.params.forEach((params: Params) => {
         let id = +params['id'];
         this.getPerson(id).then(person => {
           this.person = person;
         });
       });
  }
```

We have some dummy data at `app/data/people.json`. This is the same data used in the `members` component. We will retrieve the data based on the selected ID, like this:

```
getData() {
    return this.http.get('app/data/people.json')
      .toPromise()
      .then(response => response.json());
  }
```

The `getData()` method will retrieve the data from the API and will return a promise:

```
getPerson(id:number) {
    return this.getData().then(data => data.find(member =>
    member.id === id));
  }
```

The `getPerson()` method will resolve the returned promise and will return the `Person` object based on the selected ID.

The complete code regarding `PersonComponent` will be as follows:

```
import { Component, OnInit } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Router, ActivatedRoute, Params } from '@angular/router';
import 'rxjs/add/operator/toPromise';

@Component({
  selector: 'app-person',
  moduleId: module.id,
  templateUrl: 'person.component.html',
  styleUrls: ['../members.component.css']
})
export class PersonComponent implements OnInit {
  person: Person;
  constructor(private http:Http, private route: ActivatedRoute, private
router: Router) {
  }

  ngOnInit() {
    this.route.params.forEach((params: Params) => {
```

```
        let id = +params['id'];
        this.getPerson(id).then(person => {
          this.person = person;
        });
      });
  }

  getPerson(id:number) {
    return this.getData().then(data => data.find(member => member.id ===
id));
  }

  getData() {
    return this.http.get('app/data/people.json')
      .toPromise()
      .then(response => response.json());
  }
}
```

The `search` component template contains the search form and the list of search results when there are some results to show.

The template looks as follows:

```
<h2>Member Details</h2>

<table *ngIf="person" id="personDetails">
  <tbody>
  <tr>
    <td>Name :</td>
    <td>{{person.name}}</td>
  </tr>
    <tr>
      <td>Phone: </td>
      <td>{{person.phone}}</td>
    </tr>
    <tr>
      <td>Street: </td>
      <td>{{person.address.street}}</td>
    </tr>
    <tr>
      <td>City: </td>
      <td>{{person.address.city}}</td>
    </tr>
    <tr>
      <td>State: </td>
      <td>{{person.address.state}}</td>
    </tr>
```

```
      <tr>
        <td>Zip: </td>
        <td>{{person.address.zip}}</td>
    </tr>
    </tbody>
  </table>
```

# Search results in the route

We have the search result/`Person` component, but we forgot to include that in the router configuration. Without that, we will have an exception as it will not be possible to navigate to the `Person` component from the `members` list without having it in the route.

So, in our existing `app.routes.ts` file, we will add the following search routes:

```
export const rootRouterConfig: Routes = [
  {
    path: '/person/:id',
    component: PersonComponent
  }
.................
];
```

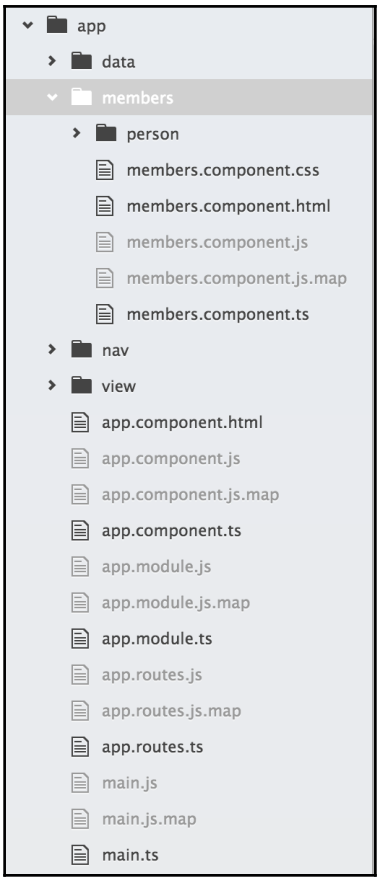
# Running the search wheel

Our application is ready with the restructure, route configuration, e2e testing, and components with their child components. We will look at the current file structure and output of the project.

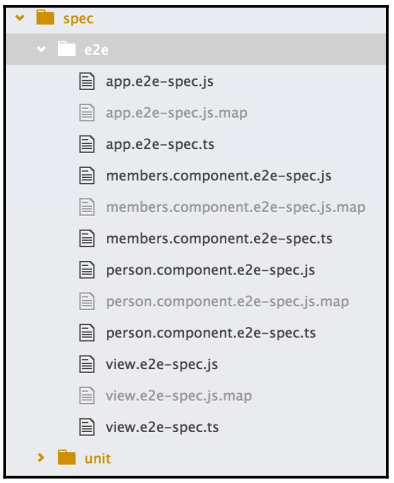
# App structure

We have two major folders in our application, one is the `app` directory and the other is the `spec/test` directory.

Let's see the current structure of our `app` directory:



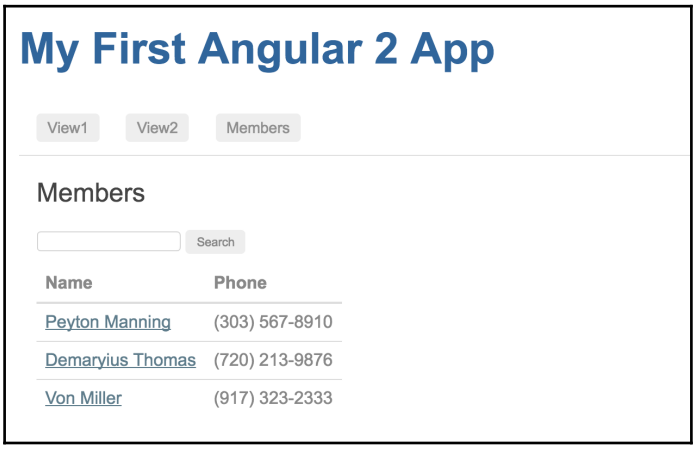
And here's the `test` directory:



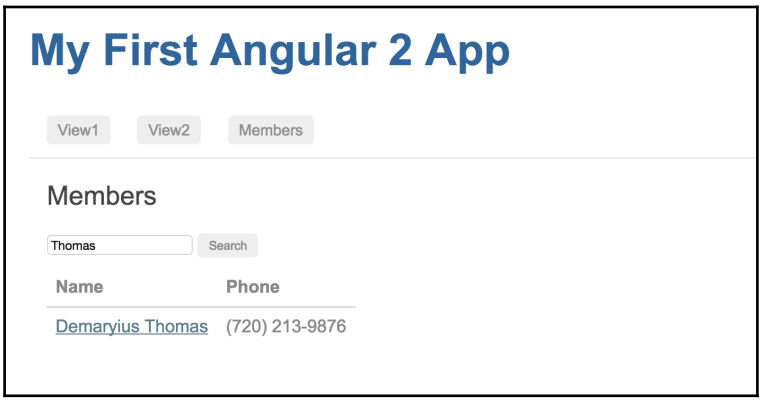

# Let's run

Our search feature is ready to run. If we run `npm start`, our application will run in the browser on the port `3000` by default. Let's navigate to **Members** to get the output of the search feature. The search feature URL is `http://localhost:3000/members`.

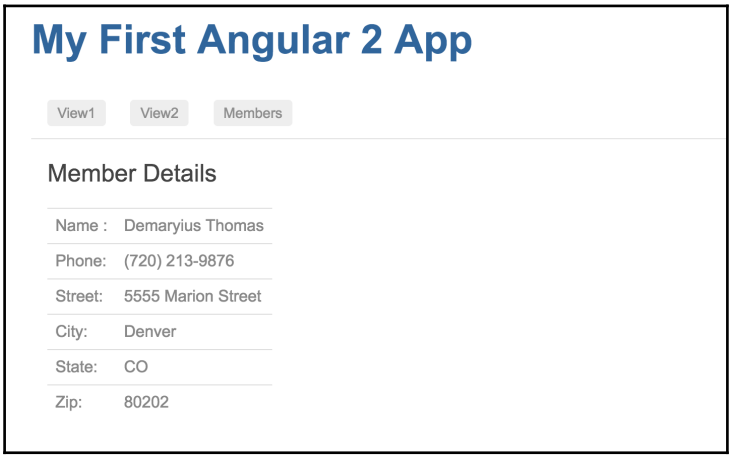
When we land on the **Members** page, it will actually load all the data as the search input is empty, which means there is no search query. The output should look as follows:



Now let's check the **Members** page with a search query. If we type `Thomas` as the query and search, it will give us only one data row, as shown here:




We have one row in the data list. Now it's time to see the details of the data. After clicking on **Thomas**, we will see detailed information about Thomas, with the address, as follows:




Hurray! The complete application is running in the browser as expected.

# How's e2e now!

The project is running in the browser, and we've done e2e testing for every component. Let's see how the e2e test reacts when we run the whole application's e2e test together.

Let's run `npm run e2e`; the output is as shown here:

```
[1] [23:48:38] I/direct - Using ChromeDriver directly...
[1] [23:48:38] I/launcher - Running 1 instances of WebDriver
[1] AppDir: ./
[1] Started
[1]
[1]   Suite: Given should test the search feature
[1] .
[1]     passed - should have an input and search button
[1] .
[1]     passed - There should be one item in search result
[1]   Suite passed: Given should test the search feature
[1]   Suite: Given should test the search result in details view
[1] .
[1]     passed - should be load the person details page
[1]   Suite passed: Given should test the search result in details view
[1]   Suite: Given views should flip through navigation in
[1] .
[1]     passed - View1 should load and visible initially
[1] .    passed - View1 should have body content as expected
[1] .
[1]     passed - Should flipped to View2 and view2 should visible
[1] .
[1]     passed - Should flipped to View2 and should have body content as expected
[1] .
[1]     passed - Should flipped to View1 again and should visible
[1]   Suite passed: Given views should flip through navigation in
[1]
[1]
[1]
[1] 8 specs, 0 failures
[1] Finished in 17.085 seconds
[1]
[1]
[1] [23:49:00] I/launcher - 0 instance(s) of WebDriver still running
[1] [23:49:00] I/launcher - chrome #01 passed
```

# Self-test questions

Q1. Which custom placeholder is used to load the component after navigation?

```
<router-output> </router-output>

<router-outlet> </router-outlet>

<router-link> </router-link>
```

Q2. Given the following Angular component, how would you select the `element` and simulate a click?

```
<a href="#">Some Link</a>
$('a').click();.
element(by.css('li')).click();.
element(by.linkText('Some Link')).click();.
```

Q3. When using routes with Angular, you need to install `@angular/router`.

- True
- False

# Summary

This chapter showed us how to use TDD to build an Angular application. The approach, up to this point, has focused on the specification from a user perspective and using TDD with a top-down approach. This technique helps us get usable and small components tested and completed for the users.

As applications grow, so does their complexity. In the next chapter, we will explore the bottom-up approach and see when to use that technique over the top-down approach.

This chapter showed us how TDD can be used to develop a component-based application with navigation by routers. Routes allow us to get a nice separation of our components and views. We looked at the usage of several Protractor locators, from CSS to repeaters, link text, and inner locators. Besides using Protractor, we also learned how to configure Karma with a headless browser, and we got to see it in action.

# 8
# Telling the World

The build up of TDD focused on fundamental components, namely the life cycle and process, using step-by-step walk-throughs. We have studied several applications from the ground up, understanding how to build Angular applications and use tools to test them.

It's time to expand further into the depths of Angular and integrate services, EventEmitters, and routes.

This chapter will be slightly different from the others in a few ways:

- Instead of building a brand new application, we will use the search application from `Chapter 7`, *Flip Flop*
- We will add the unit tests for Angular routes and navigation that were skipped in previous chapters
- We will make the existing search application more modern by separating the commonly used actions into services
- We will take advantage of the Angular `EventEmitter` class to communicate between the different components

## Getting ready to communicate

We will follow a different approach in this chapter, as we've already learned the TDD approach. We developed a small project in the previous chapter, and our plan is to work with that project and make it better in order to present it to the world.

So, before the walk-through, we will have to review and identify any problems and the scope for improvement of the project. To do so, we have to be confident of the code base of the search application.

# Loading the existing project

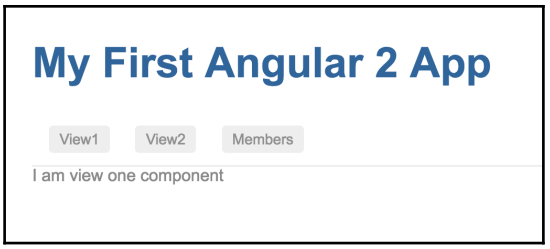
To start with, we will copy the project from `Chapter 7`, *Flip Flop*, which was originally from `https://github.com/angular/quickstart`, and rename it `angular-member-search`.

Let's proceed and get ready to run it:

```
$ cd angular-member-search
$ npm install
$ npm start
```

To confirm the installation and run the project, the application will automatically run it in a web browser.

Here is the output we should get when we will run the project:




Oh! We have our end-to-end test ready in the project. Before we go for an update, we have to be sure that the existing e2e tests are successful.

Let's run the `e2e` test in a separate console:

```
$ npm run e2e
```

Yes, everything passes successfully:

```
[1] [23:48:38] I/direct - Using ChromeDriver directly...
[1] [23:48:38] I/launcher - Running 1 instances of WebDriver
[1] AppDir: ./
[1] Started
[1]
[1]   Suite: Given should test the search feature
[1] .
[1]     passed - should have an input and search button
[1] .
[1]     passed - There should be one item in search result
[1]   Suite passed: Given should test the search feature
[1]   Suite: Given should test the search result in details view
[1] .
[1]     passed - should be load the person details page
[1]   Suite passed: Given should test the search result in details view
[1]   Suite: Given views should flip through navigation in
[1] .
[1]     passed - View1 should load and visible initially
[1] .    passed - View1 should have body content as expected
[1] .
[1]     passed - Should flipped to View2 and view2 should visible
[1] .
[1]     passed - Should flipped to View2 and should have body content as expected
[1] .
[1]     passed - Should flipped to View1 again and should visible
[1]   Suite passed: Given views should flip through navigation in
[1]
[1]
[1]
[1] 8 specs, 0 failures
[1] Finished in 17.085 seconds
[1]
[1]
[1] [23:49:00] I/launcher - 0 instance(s) of WebDriver still running
[1] [23:49:00] I/launcher - chrome #01 passed
```

# Unit testing

In the previous chapter, we started with the top-down approach. The goal was to elaborate on end-to-end testing based on what we had learned. We had the user scenario clear, we went through the tests, and our scenario passed our implementation.

In the previous chapter, we only covered end-to-end testing. So, in this chapter, we'll cover unit tests as much as we can.

Also, in the previous chapter, we mostly looked at Angular routes and navigation. So now, as a logical extension, we will look at how to test Angular routes and navigation.

# Testing a component

Before we go ahead with the component test, we should discuss some points about testing Angular components. We already have a basic idea: in Angular, everything is a combination of some components. So it would be great for us to learn in more detail about Angular component testing.

We can test a component in various ways, based on its behavior and use case. We could even have test specs for multiple components when they work together as an application.

Let's have a look at some of the ways of testing components.

# Isolated testing

Isolated testing, also known as solo testing, is named so because this type of test can run without the need to compile components according to test specs. If it doesn't compile, it will not have the compiled template in the test spec; only the component class and its methods. This means that if a component's features are not very DOM dependent, it can be tested in an isolated manner.

Isolated testing is mostly used for complex feature or calculation testing, where it just initiates the component class and calls all the methods.

For example, take a look at the unit tests of `Chapter 6`, *The First Step*, where `AppComponent` was responsible for adding comments and increasing their likes:

```
beforeEach(() => {
    comp = new AppComponent();
    comp.add('a sample comment');
    comp.like(comp.comments[0]);
});

    it('First item in the item should match', () => {
        expect(comp.comments[0].title).toBe('a sample
        comment');
    });

    it('Number of likes should increase on like', () => {
        expect(comp.comments[0].likes).toEqual(1);
    });
```

# Shallow testing

Isolated testing sometimes fulfills the requirements of the test spec, but not always. Most of the time, components have DOM dependent features. In such cases, it is important to render the component's template in the test specs so that we have the compiled template in the scope and test specs are able to interact with DOM.

For example, if we want to write a basic unit test for our `AppComponent`, which is mostly DOM dependent as there is no method in the component class, then we just need to compile the component and check that it is defined. In addition, we can have a test spec if the component's template has the correct text inside the `<h1>` element.

The code will look as follows:

```
beforeEach(async(() => {
    TestBed.configureTestingModule({
        declarations: [ AppComponent ]
    })
    .compileComponents();
}));

beforeEach(() => {
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
    de = fixture.debugElement.query(By.css('h1'));
});

it('should create and initiate the App component', () => {
    expect(comp).toBeDefined();
});

it('should have expected test in <h1> element', () => {
    fixture.detectChanges();
    const h1 = de.nativeElement;
    expect(h1.innerText).toMatch(/My First Angular 2 App/i,
    '<h1> should say something about "Angular App"');
});
```

# Integration testing

The following are some key points on integration testing:

- The name *integration testing* should give us some idea of what kind of test it is. It is similar to shallow testing as it also needs to compile the component with the template and interact with the DOM.
- We will next look at our route and navigation test suite, where we will integrate the `AppComponent`, router, and navigation test suites.
- We have a test suite ready for `AppComponent`, as it includes the `navbar` component and `router-outlet` component. All of these together work to fulfill the routing specs.
- So, to get a confident test spec for a router, we should go with integration testing.

We will look at an example of router testing with a detailed explanation in the following sections.

The main difference between integration testing and shallow testing is that integration testing works for the test suite of the complete application, or a small portion of the application, where multiple components work together to solve some purpose. It shares some similarities with end-to-end testing, but with a different approach.

# Karma configuration

In previous chapters, the default Karma configuration was used, but no explanation about this default configuration has been given yet. **File watching** is a useful default behavior that will now be reviewed.

# File watching

File watching is enabled by default when the Karma `init` command is used. File watching in Karma is configured with the following definition in the `karma.conf.js` file:

```
autoWatch: true,
```

The file watching feature works as expected and watches the files defined in the configuration's `files` array. When a file is updated, changed, or deleted, Karma responds by rerunning the tests. From a TDD perspective, this is a great feature, as tests will continue to run without any manual intervention.

The main point to watch out for is the addition of files. If the file being added doesn't match the criteria in the `files` array, the `autoWatch` parameter won't respond to the change. As an example, let's consider that the files are defined as follows:

```
files : [ 'dir1/**/*.js']
```

If this is the case, the watcher will find all the files and subdirectory files ending in `.js`. If a new file is in a different directory, not in `dir1`, then the watcher will not be able to respond to the new file because it is in a different directory from what it was configured in.

# Testing routers and navigation

We were introduced to Angular routers and navigation alongside the general components in `Chapter 7`, *Flip Flop*.

As we have discussed the different types of tests for Angular components, routers and navigation, we will look at integration testing. For that, we will use our application component test, that is, our base component, and we will then integrate navigation and `router-outlet` component tests with the application component to test the router.

# Testing the app component

Before we go ahead with router testing, we will get ready with our application component tests. In the app component test, we will test whether the component is defined and initiated correctly, and then we will test the page title by selecting the DOM element.

We learned about shallow testing in previous sections; when we interact with DOM elements, we need shallow testing. The same goes here: as we will have to deal with DOM elements, we will use shallow testing as our application component test.

For shallow testing, we will need to depend on the `TestBed` Angular test API from Angular core testing, which will be used to compile and initiate the components in the test suite. Besides that, we will have to depend on the `ComponentFixture` module from core testing. We will need two more modules, named `By` and `DebugElement`, from the Angular core and platform APIs to interact with DOM elements.

Our app component test will be located at `spec/unit/app.component.ts` and will look like this:

```
import { async, ComponentFixture, TestBed } from '@angular/core/testing';
import { By }           from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

import { AppComponent } from '../../app/app.component';
import { NavbarComponent } from '../../app/nav/navbar.component';
import { RouterOutlet } from '@angular/router';

describe('AppComponent test suite', function () {
  let comp: AppComponent;
  let fixture: ComponentFixture<AppComponent>;
  let de: DebugElement;

  beforeEach(async(() => {
```

```
    TestBed.configureTestingModule({
       declarations: [ AppComponent ]
     })
     .compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
    de = fixture.debugElement.query(By.css('h1'));
  });

  it('should create and initiate the App component', () => {
    expect(comp).toBeDefined();
  });

  it('should have expected test in <h1> element', () => {
    fixture.detectChanges();
    const h1 = de.nativeElement;
    expect(h1.innerText).toMatch(/My First Angular 2 App/i,
      '<h1> should say something about "Angular App"');
  });
});
```

If we run this test, we will see the following result:

```
Chrome 54.0.2840 (Mac OS X 10.10.5): Executed 2 of 2 SUCCESS
(0 secs / 0.522 secs)
```

Our application component test is ready now; next, we will perform a router test, including `router-outlet` and navigation.

# Testing router

The Angular router is not part of Angular core; it's a separate module that has to be imported before being used. It has some directives, such as `RouterOutlet` and `RouterLink`, which play an active role in fulfilling router activities. To test the router, first we will test these directives, in order to prepare the platform for testing the complete router.


We can test the router using the actual router module, but sometimes it creates some complexity for the entire routing system. Due to this, test specs may fail without providing an accurate error. To avoid this, it's recommended to create router stubs and use those for router testing.

# Router stubs

I had the router stubs idea from Angular's official testing docs. I liked the idea about the routing stubs and copied the `router-stubs` file from `angular.io/public/docs/_examples/testing/ts/testing/router-stubs.ts` in Angular's GitHub repository. The first router stubs directive is `RouterStubLinksDirective`, which is responsible for hosting the element or anchor link (`<a>`) to perform the `click` event for the directive's `onClick()` method. The URL bound to the `[routerLink]` attribute flows to the directive's `linkParams` property. When the anchor link (`<a>`) is clicked on, it should trigger the `onClick()` method, and it will set to the tentative `navigateTo` property.

This `router-stubs` file has a dependency on the Angular router and relevant directives, including `RouterLink` and `RouterOutlet`, so we will need to import those.

So, the router stubs will be located at `spec/unit/stub/router-stub.ts`, and the code will be as follows:

```
export  {Router, NavigationExtras, RouterLink, RouterOutlet} from
'@angular/router';

import { Component, Directive, Injectable, Input } from '@angular/core';

@Directive({
  selector: '[routerLink]',
  host: {
    '(click)': 'onClick()'
  }
})
export class RouterLinkStubDirective {
  @Input('routerLink') linkParams: any;
  navigatedTo: any = null;

  onClick() {
    this.navigatedTo = this.linkParams;
  }
}
```

Beside the `RouterLinkStubDirective`, this stub should contain the `RouterOutletStubComponent` to support the `router-outlet` directive, and `RouterStub` to support the main router module:

```
@Component({selector: 'router-outlet', template: ''})
export class RouterOutletStubComponent { }
```

```
@Injectable()
export class RouterStub {
  navigate(commands: any[], extras?: NavigationExtras) { }
}
```

## The router-outlet and navigation test

As we know, the `router-outlet` and navigation (`RouterLink`) menus work together with the application landing page, that is, our application component. The testing mechanism will be of the same form. This means that we will test both of these modules with the application component.

As mentioned a bit earlier, we will use an integration test here for `router-outlet` testing. We have our application component test suite ready; now it's time to integrate `router-outlet` and navigation (`RouterLink`), and we will have our integration test suite for the application component along with `router-outlet` and `RouterLink`.

We have the `navbar` component, which is basically a navigation component that contains the `RouterLink` to navigate through the router. We will have to import that component to our test suite for it to perform correctly. Besides the actual router module, we will have to import the `RouterStub` that we have created. To reiterate, `router-stubs` contains the `RouterOutletStubComponent` and `RouterLinkStubDirective` components.

After importing all the required components, we will have to declare them in the `TestBed` configuration. And, as a part of the setup, we will get all the `navLinks` from the `RouterLinkStubDirective` in the test suite's scope to test and bind `click` events to the `linkParams`.

The test suite's setup will look like this:

```
import { NavbarComponent } from '../../app/nav/navbar.component';
import { AppComponent } from '../../app/app.component';
import { RouterOutletStubComponent, RouterLinkStubDirective } from
'./stub/router-stubs.js';

describe('AppComponent test suite', function () {
  let navDestination:any;
  let navLinks:any;
  let fixture: ComponentFixture<AppComponent>;
  let de: DebugElement;

  beforeEach(async(() => {
   TestBed.configureTestingModule({
      declarations: [ AppComponent, NavbarComponent,
```

```
                    RouterLinkStubDirective,
                    RouterOutletStubComponent ]
  })
  .compileComponents();
}));

beforeEach(() => {
  fixture.detectChanges();

  navDestination = fixture.debugElement
    .queryAll(By.directive(RouterLinkStubDirective));

  navLinks = navDestination
    .map(de => de.injector.get(RouterLinkStubDirective) as
    RouterLinkStubDirective);
});
```

For the test specs, first we will test the link parameters in the navigation menu. We have the `navLinks`, and we will match them with the `linkParams` of the `navLinks`.

Then, we will test the expected navigation while clicking on the navigation menu items. We will test that with the help of the `navigatedTo` method.

Our test specs will look as follows:

```
it('can get RouterLinks from template', () => {
  expect(navLinks.length).toBe(3, 'should have 3 links');
  expect(navLinks[0].linkParams).toBe('/view1', '1st link should
  go to View1');
  expect(navLinks[1].linkParams).toBe('/view2', '1st link should
  go to View2');
  expect(navLinks[2].linkParams).toBe('/members', '1st link should
  go to members search page');
});

it('can click nav items link in template and navigate
accordingly', () => {
  navDestination[0].triggerEventHandler('click', null);
  fixture.detectChanges();
  expect(navLinks[0].navigatedTo).toBe('/view1');

  navDestination[1].triggerEventHandler('click', null);
  fixture.detectChanges();
  expect(navLinks[1].navigatedTo).toBe('/view2');

  navDestination[2].triggerEventHandler('click', null);
  fixture.detectChanges();
  expect(navLinks[2].navigatedTo).toBe('/members');
```

```
});
```

So, we can say that this will cover the tests for `router-outlet` and routerLink, which will confirm that the router links are working as expected, and that we are able to navigate though the expected router after clicking on the navigation menu.

# Implementing an integration test

Our test specs are ready. We have been planning for an integrated test, and we can perform one now. Here, we are combining the app component and `navbar` component, along with `router-outlet` and `routerLink`, to test the route and navigation. We have to interact with DOM elements with the help of the `debugElement` module from the browser platform API.

The test suite is ready–time to run the test.

Let's run it with the following command:

```
npm test
```

And all the test specs pass as expected. The result will be as follows:

```
Chrome 54.0.2840 (Mac OS X 10.10.5): Executed 4 of 4 SUCCESS
(0 secs / 1.022 secs)
```

# More tests…

We just added some tests that will cover a few of the features that we have developed so far, mostly focused on the router (`router-outlet` and `routerLink`).

We will add more tests for members and the search feature, but we will update the behavior of the existing features of searching and member listing. Besides that, our current code base does not have proper decoupling between the components' features, which will make it complex to test features individually.

We already have the end-to-end test, which will verify the output we expect from our components, but for unit testing, we will need to refactor the code and decouple them. We will cover the tests for the rest of the features after we update the behavior and refactor the correct code base.

# Recap of the application behavior

Let's look at a quick overview of the search application:

- Our search application invokes the Members component in the DOM. It contains two major parts: the search area and the result area.
- From the search area, we type a search query and submit it to get the expected result in the result area.
- The resulting area lists down the member list based on the search query. We may have figured out that we get all the data during the initialization of the `Members` component; that's because we call the Members component's `search()` method with `ngOnInit()`, and it returns all the data as our logic has been set to return all data when no search query is set.
- By clicking on a member's name, we can see the detailed information about that member on the details page.

# Updating the application behavior

According to the previous specification, it seems we have some incorrect behavior in the search feature. Right now, we are calling `search()` when initializing the members of the search component. This seems a bit wrong; we should start the search after entering the search query and/or clicking on the **Search** button.

The expected behavior is that it will first load all the member data and then, after starting the search, the data list will be updated based on the search query.

To do so, let's update the `ngOnInit()` method in `members.component.ts` and add a new method, `getMember()`, to have the entire data list during component initiation.

The expected change will be as follows:

```
ngOnInit() {
    this.getMembers();
  }

  getMembers() {
    this.getData()
    .then(data => {
      data.map(item => {
        this.memberList.push(item);
      });
    })
```

```
      return this.memberList;
    }

  search() {
      // Do Search
    }
```

# Identifying the problem

Based on the existing code, it seems we have defined the `getData()` method twice, in `members.component.ts` and `person.component.ts`, because in both the components, we had to call the JSON data source to get the member dataset.

So, what's the problem with that? It's bad practice as it duplicates code, and duplication of code is hard to manage when the application becomes large and complex.

For example, now we have the following method twice:

```
getData() {
    return this.http.get('app/data/people.json')
      .toPromise()
      .then(response => response.json());
  }
```

If we have to to change the data source URL or API, we will have to change this method in two places. It's not so tough to change it twice, but what about 10-12 times, or even more for a larger application?

Yes, it's a problem, and it needs a solution.

# Finding a solution

We've identified the problem, which is code duplication. We know the solution: we have to write the method in a common place and use it in both components. In short, we have to make this method reusable so that every component can share it.

It seems simple, but we have to do it in the Angular way. We can't just move the method to a separate file and import that.

Angular introduced services for such situations. We'll now look at some of those services with examples.

# Angular services

Angular services were introduced to write code shareable among components. So if we need a piece of code for many components, it's recommended to create a single reusable service, and wherever we need that piece of code, we can just inject that service to the component and use its methods as needed.

Services are used to abstract application logic. They are used to provide a single responsibility for a particular action. Single responsibility allows components to be easily tested and changed. This is because the focus is on one component and not all the inner dependencies.

Mostly, a service acts as the data source of any application. Whenever we need a piece of code to communicate with the server to get data (mostly JSON), we use a service.

This is because most components need to access data, and everyone can inject the common service as required. So, we have a commonly used piece of code, which is actually the data layer for our application. We should move those parts to a service to make our application smart so that we can tell the world we are not duplicating code in any way.

# We have service now?

As planned, we have moved the `getData()` method from the `members.component.ts` and `person.component.ts` components to a new file so that we can get rid of code duplication.

Let's create a new file at `app/services/members.service.ts`, make a new class to export, called `MembersService`, and move the `getData()` method there. Besides moving the method, we will have to import `{ Http, Response }` from the Angular HTTP module as `getData` has a dependency on HTTP.

Observe the following code sample:

```
import { Http, Response } from '@angular/http';

export class MembersService {
  constructor(private http:Http) {

  }

  getAPIData() {
    return this.http.get('app/data/people.json');
  }
```

```
  getData() {
    return this.getAPIData()
      .toPromise()
      .then(response => response.json());
  }

}
```

We have a service now, and we can start using it. Let's import and use it in the Members component.

Wait; before that, we will have to import the service into the application module to have identification of it. As long as it's a service, we will have to identify it as a provider; the service will act as a service provider.

Our `app.module.ts` file will look like this:

```
import {MembersService} from './services/members.service';

@NgModule({
  declarations: [AppComponent, NavbarComponent, ....],
  imports     : [BrowserModule, FormsModule, ......],
  providers   : [MembersService],
  bootstrap   : [AppComponent]
})
```

Now, to use the service in our component, we have to import and inject it into our MembersComponents with the service name `MembersService`. As long as we inject the service as the constructor of the component, we will have the service available to the entire component. To access the method, we need to call it `this.membersService.getData()`.

So, our Members component will look like this:

```
import { MembersService } from '../services/members.service';
@Component({
   ............
})
export class MembersComponent implements OnInit {
 constructor(public membersService: MembersService, private router:Router)
{

  }

  getMembers() {
    this.membersService.getData()
    .then(data => {
```

```
        data.map(item => {
          this.memberList.push(item);
        });
      })
      return this.memberList;
    }
```

Time to run and look at the output and see how the service works with the Members component.

Lets point the browser to `http://localhost:3000/members`.

Oops! What happened? We are getting an error in the browser console:

```
Error: (SystemJS) Can't resolve all parameters for MembersService: (?)
```

Based on the error, we have made a mistake: `SystemJS` (used as the module loaded) can't inject `MembersService` as we missed adding something to the service to make it perfect. In Angular, we have to mention in every service whether it will be injectable; without doing so, we will not be able to inject that service into any component.

And, for that, we will have to use the Angular **Injectable** decorator. We will take a look at it in brief.

# Injectable services

The Injectable decorator is a part of the Angular core library, used when creating injectable services. Without defining it as injectable, it's not possible to identify the dependency of a service. To define it as injectable, we will have to use `@Injectable()` at the top of the class definition.

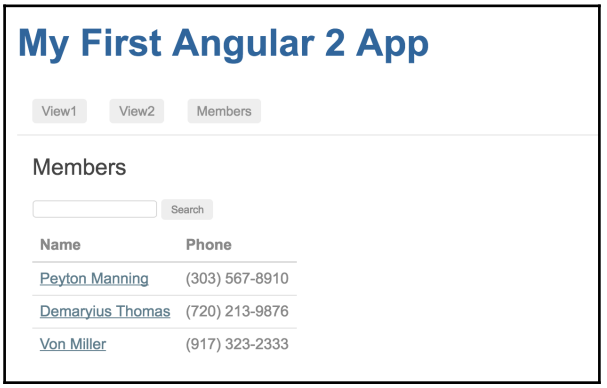
The code will look like this:

```
import { Injectable } from '@angular/core';
import { Http, Response } from '@angular/http';

@Injectable()
export class MembersService {
  constructor(private http:Http) {

  }

  getData() {
    return this.http.get('app/data/people.json')
```

```
        .toPromise()
        .then(response => response.json());
    }
}
```

We have made the service injectable. Now, we should be fine to inject it into the Members component and point our browser to `http://localhost:3000/members`.

Hurray! No more errors, and we are getting the expected data list:




Seems like our service is injectable and working fine. It's time to implement it in the `PersonComponent`, as we need the data service on that component as well. The same as the Members component, let's import and inject it into the `PersonComponent` with the service name `membersService`. Again, we will have to access the data service method with `this.membersService.getData()`.

Our `PersonComponent` will look like this:

```
import { MembersService } from '../../services/members.service';

@Component({
  ..........
})
export class PersonComponent implements OnInit {
  constructor(public membersService: MembersService, private route:
ActivatedRoute, private router: Router) {

  }

.................

  getPerson(id:number) {
```

```
        return this.membersService.getData()
              .then(data => data.find(member => member.id === id));
    }
}
```

Time to run and look at the output of how the service is working with the Members component.

We have our e2e test, which will confirm that everything is going well with the new changes:

```
$ npm run e2e
```

Yes, everything passes successfully:

```
[1] [23:48:38] I/direct - Using ChromeDriver directly...
[1] [23:48:38] I/launcher - Running 1 instances of WebDriver
[1] AppDir: ./
[1] Started
[1]
[1]   Suite: Given should test the search feature
[1] .
[1]     passed - should have an input and search button
[1] .
[1]     passed - There should be one item in search result
[1]   Suite passed: Given should test the search feature
[1]   Suite: Given should test the search result in details view
[1] .
[1]     passed - should be load the person details page
[1]   Suite passed: Given should test the search result in details view
[1]   Suite: Given views should flip through navigation in
[1] .
[1]     passed - View1 should load and visible initially
[1] .    passed - View1 should have body content as expected
[1] .
[1]     passed - Should flipped to View2 and view2 should visible
[1] .
[1]     passed - Should flipped to View2 and should have body content as expected
[1] .
[1]     passed - Should flipped to View1 again and should visible
[1]   Suite passed: Given views should flip through navigation in
[1]
[1]
[1]
[1] 8 specs, 0 failures
[1] Finished in 17.085 seconds
[1]
[1]
[1] [23:49:00] I/launcher - 0 instance(s) of WebDriver still running
[1] [23:49:00] I/launcher - chrome #01 passed
```

Yay! Our code refactoring hasn't affected our expected behavior.

# Services will serve you more

To reap the complete benefits of services, we will move two more methods from the Members and Person components. Before that, those methods were component specific; now, by adding them to a service, those methods can be used from any component just by injecting the service.

Perhaps we will benefit later from this change but want to keep these methods decoupled from the components.

The newly added code will look like this:

```
@Injectable()
export class MembersService {
  constructor(private http:Http) {

  }

  ............

  searchQuery(q:string) {
    if (!q || q === '*') {
      q = '';
    } else {
      q = q.toLowerCase();
    }
    return this.getData()
      .then(data => {
      let results:any = [];
      data.map(item => {
        if (JSON.stringify(item).toLowerCase().includes(q)) {
          results.push(item);
        }
      });
      return results;
    });
  }

  getPerson(id:number) {
    return this.getData()
   .then(data => data.find(member => member.id === id));
  }
}
```

# Testing the service

The goal behind the code decoupling and separation was to make the code testable. We did so, and we have separated the data retrieval part from the Members component and made a service so that it will be easy to test. The service is injectable; other than that, it's similar to an angular component. So, to perform unit testing, we will test the methods that the service contains.

# Testing service injection

Like other Angular components, we can test whether the service is defined well. But the main difference is that, as long as the service is injectable, we will need to inject it in the test specs to get the instance to test.

For a sample test spec, we can set it up so that it will import the `TestBed` and `inject`, and then configure the `TestingModule` using MembersService as the provider. Then, in the test spec, we will inject the service and check whether the service is defined as expected.

Our sample test suite will look like this:

```
import { inject, TestBed } from '@angular/core/testing';
import { MembersService } from '../../app/services/members.service';

describe('Given service should be defined', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        MembersService,
      ],
    });
  });

 it('should initiate the member service', inject([MembersService],
(membersService) => {
    expect(membersService).toBeDefined();
  }));

});
```

For this test, the expected result will be true.

# Testing HTTP requests

To have a unit test for HTTP requests, we will have to use the async technique to keep the HTTP call asynchronous, and in Angular testing, we will use the `fakeAsync` module, which is an async module to use with mock HTTP requests.

Wait, “mock”?

Well yes; to test HTTP requests in an Angular test suite, we don't need to make actual HTTP requests. To achieve the effect of an HTTP request, we can mock out our HTTP services; Angular has provided a mock service called **MockBackend**.

`MockBackend` is a class that can be configured to provide mock responses for HTTP mock requests, and it will work exactly the same as HTTP services but without making the actual network requests.

After we have configured the `MockBackend`, it can be injected into HTTP. So, from our service where we used `http.get`, we'll have the expected data returned.

Our test suite with the HTTP request will look like this:

```
import { fakeAsync, inject, TestBed } from '@angular/core/testing';

import { Http, BaseRequestOptions, Response, ResponseOptions } from
'@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';

import { MembersService } from '../../app/services/members.service';

const mockData = {
  "id": 2,
  "name": "Demaryius Thomas",
  "phone": "(720) 213-9876",
  "address": {
    "street": "5555 Marion Street",
    "city": "Denver",
    "state": "CO",
    "zip": "80202"
  }
};

describe('Given service should be defined and response HTTP request', () =>
{

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        MembersService,
        BaseRequestOptions,
        MockBackend,
        {
          provide: Http,
          useFactory: (backend, defaultOptions) => {
            return new Http(backend, defaultOptions);
          },
          deps: [MockBackend, BaseRequestOptions],
        },
      ],
    });
```

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

Here, at first, besides importing the `MockBackend`, we are importing the `MockConnection`, which is used to subscribe to the backend connection and provide the connected data to the next steps. Then, we configure the `MockBackend`, which will return the HTTP object.

Next, we will get ready with our test specs by injecting `MockBackend` and `MembersService`:

```
    it('should return response when subscribed to getUsers', fakeAsync(
      inject([MockBackend, MembersService], (backend, membersService) => {
        backend.connections.subscribe(
          (c: MockConnection) => {
            c.mockRespond(
              new Response(
                new ResponseOptions({ body: mockData })
              ));
            });

          membersService.getAPIData().subscribe(res => {
            expect(res.json()).toEqual(mockData);
          });
    })));

  });
```

In the test spec, we have injected `MockBackend`, in addition to the MembersService. `MockBackend` will subscribe to the `backend` service with the MockConnection object. MockConnection will create a new `ResponseOptions` object, where, with a `ResponseOptions` object we can configure our response properties.

Here, we just set the `body` property of the response object and set the `body` value to a predefined `mockData` object.

# Service stubs

We can test the service with stub data as well. For example, we can make a stubbed version of our `MembersService` called `MembersServiceSpy`, which will fake all the necessary features of that service.

This fake service will return a resolved `Promise` with mock data, so we can just use this stubbed method for testing. It will create a spy for all of the methods we have in the service and return a separate `Promise` for every single method.

The stubbed service will be located at `spec/unit/stub/members.service.stub.ts`, and it will be as follows:

```
import { Component, Directive, Injectable, Input } from '@angular/core';

export class MembersServiceSpy {
  members = {
    "id": 2,
    "name": "Demaryius Thomas",
    "phone": "(720) 213-9876",
    "address": {
      "street": "5555 Marion Street",
      "city": "Denver",
      "state": "CO",
      "zip": "80202"
    }
  };

  getData = jasmine.createSpy('getData').and.callFake(
    () => Promise
      .resolve(true)
      .then(() => Object.assign({}, this.members))
  );

  getPerson = jasmine.createSpy('getPerson').and.callFake(
    () => Promise
      .resolve(true)
      .then(() => Object.assign({}, this.members))
  );

  searchQuery = jasmine.createSpy('searchQuery').and.callFake(
    () => Promise
      .resolve(true)
      .then(() => Object.assign({}, this.members))
  );

}
```

## Service test with stubbed data

Here, we will test the `MembersService` with stubbed data. To do so, we will need to import the stubbed service. And with the `TestBed` configuration, we will have to provide `MemberServiceSpy` as a service instead of an actual member service.

The `MembersService` test suite's code will look as follows:

```
import { MembersServiceSpy } from './stub/members.service.stub.js';
import { MembersService } from '../../app/services/members.service';

const mockData = {
  "id": 2,
  "name": "Demaryius Thomas",
  "phone": "(720) 213-9876",
  "address": {
    "street": "5555 Marion Street",
    "city": "Denver",
    "state": "CO",
    "zip": "80202"
  }
};

describe('Given service will response for every method', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [{ provide: MembersService, useClass: MembersServiceSpy }]
    });
  });

  it('should return data', fakeAsync(inject(
    [MembersService], (service) => {
      service.getData();
      expect(service.members).toEqual(mockData);
    })));

    it('should return data', fakeAsync(inject(
      [MembersService], (service) => {
        service.searchQuery('Thomas');
        expect(service.members.name).toBe('Demaryius Thomas');
    })));

    it('should return data', fakeAsync(inject(
      [MembersService], (service) => {
        service.getPerson(2);
        expect(service.members.id).toBe(2);
    })));

});
```

# Combining and running the service's tests

We have two test suites for the Members service here. We can bring both together and run the test.

The full test suite's code will look like the following code snippet:

```
import { fakeAsync, inject, TestBed } from '@angular/core/testing';

import { Http, BaseRequestOptions, Response, ResponseOptions } from
'@angular/http';
import { MockBackend, MockConnection } from '@angular/http/testing';

import { MembersServiceSpy } from './stub/members.service.stub.js';
import { MembersService } from '../../app/services/members.service';

const mockData = {
  "id": 2,
  "name": "Demaryius Thomas",
  "phone": "(720) 213-9876",
  "address": {
    "street": "5555 Marion Street",
    "city": "Denver",
    "state": "CO",
    "zip": "80202"
  }
};

describe('Given service should be defined and response HTTP request', () =>
{

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [
        MembersService,
        BaseRequestOptions,
        MockBackend,
        {
          provide: Http,
          useFactory: (backend, defaultOptions) => {
            return new Http(backend, defaultOptions);
          },
          deps: [MockBackend, BaseRequestOptions],
        },
      ],
    });
  });
```

```
  it('should initiate the member service', inject([MembersService],
(membersService) => {
    expect(membersService).toBeDefined();
  }));

  it('should return response when send HTTP request', fakeAsync(
    inject([MockBackend, MembersService], (backend, membersService) => {
      backend.connections.subscribe(
        (c: MockConnection) => {
          c.mockRespond(
            new Response(
              new ResponseOptions({ body: mockData })
            ));
          });

        membersService.getAPIData().subscribe(res => {
          expect(res.json()).toEqual(mockData);
        });
  })));

});

describe('Given service will response for every method', () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [{ provide: MembersService, useClass: MembersServiceSpy }]
    });
  });

  it('should return data', fakeAsync(inject(
    [MembersService], (service) => {
      service.getData();
      expect(service.members).toEqual(mockData);
    })));

    it('should return data', fakeAsync(inject(
      [MembersService], (service) => {
        service.searchQuery('Thomas');
        expect(service.members.name).toBe('Demaryius Thomas');
    })));

    it('should return data', fakeAsync(inject(
      [MembersService], (service) => {
        service.getPerson(2);
        expect(service.members.id).toBe(2);
    })));
```

```
});
```

The test suite for the Members service is ready to run. Let's run it with this command:

```
npm test
```

All the test specs pass as expected. The result will be as follows:

```
Chrome 54.0.2840 (Mac OS X 10.10.5): Executed 9 of 9 SUCCESS
(0 secs / 4.542 secs)
```

# Communicating through the power of events

Angular has more powerful event-handling capabilities compared to Angular 1.x. Angular 1.x has two-way data binding, whereas Angular doesn't recommend that. Angular handles the communication between data and templates through the power of events.

Angular projects stand on the combination of some components. To function, these components need to communicate with each other to share data and events. Mostly, components need to communicate when they have a parent-child relationship. There are a few ways in which Angular can communicate between parent and child components. The best is by handling custom events. We will look at details about custom events and see how they work with our search application.

# Angular events

As we know, Angular recommends one-way data binding, which means only from components to DOM elements. This is unidirectional data flow, and it is how Angular works. What about when we need data flow in the other direction–from DOM elements to components? Doing so depends on different events, such as clicks, keystrokes, mouseover, and touch. These events will bind with DOM elements to listen to user action and pass that action to the component.

The event binding syntax consists of the target event with parentheses to the left-hand side of the equals sign. The component contains the target event as a method, so whenever the event triggers, it will call the method from the component.
Let's look at the event that we have in the search form:

```
<button type="button" (click)="search()">Search</button>
```

Any element's events are the common targets, but it's a bit different with Angular, as Angular at first checks whether the target name matches with the event property of any known directives or components.

# Custom events in Angular

Custom events are raised by directives or components with Angular `EventEmitter`. Directives create an `EventEmitter` object and expose themselves as its property to be passed via the `@Output` decorator. We will look at details of `@Output` decorator next. After exposing an `EventEmitter` object as a property, the directives will call `EventEmitter.emit(value)` to fire the event and pass the value to the parent directives.

The custom directive/component class will define the custom event as follows:

```
@Output() someCustomEvent: EventEmitter<any> = new EventEmitter();

  this.someCustomEvent.emit(value);
```

Parent directives will listen for the event by binding to this property and will receive the value through the `$event` object.

The parent directive/component will contain the custom directive as follows, where it will contain the custom event as `someCustomEvent`, which will trigger the `doSomething()` method of the parent directives:

```
<custom-component (someCustomEvent)="doSomething($event)"></custom-
component>
```

The parent directive/component will contain the `doSomething()` method, as shown here:

```
doSomething(someValue) {
    this.value = someValue;
}
```

# The Output and EventEmitter APIs

Output is a decorator class from the Angular core that is used to pass custom events from a child to a parent component. To use it, we need to import it from `@angular/core`.

When we set a custom event as `@Output`, that event will be available to listen to in the parent component. This decorator will be placed inside the class, as follows:

```
export class SearchComponent {
  @Output() someCustomEvent: EventEmitter<any> = new EventEmitter();
}
```

`EventEmitter` is also a core class of Angular. When we need to use it, we will have to import it from `@angular/core`. The `EventEmitter` API is used to notify the parent component by calling `EventEmitter.emit(value)` whenever the value changes in the child component. As we know, the parent component always listens to the custom event.

# Planning further improvements

The search application we have so far is a simple search application. But we can make it better by keeping it as simple as it is. What I mean is, we can do this in the best way, like we were trying to decouple the data logic by separating the reusable code in the new service.

We still have a few more things to improve. It seems our application has not yet decoupled perfectly. Our components have not decoupled as we'd expected. We are talking about the `MembersComponent`, which contains the search feature and member listing feature.

We will follow the single responsibility principle here, which means every component should have a single responsibility. Here, `MembersComponent` has two. So we should break down this component to two separate components.

Let's break it down to two separate components, called `MembersComponent` and `SearchComponent`. In fact, we just made a plan for a new component called `SearchComponent` and brought the search feature over there from the Members component.

Now let's make a plan for the behavior expected from both components:

- The search component will have the responsibility of taking the user input as a search query and getting the expected search result using the service we have
- We then pass the search result to the Members component
- The Members component will get the search result from the search component and will bind the data list to the DOM
- Both components will communicate and exchange data using events

The plan is to make this simple app perfect by following best practices and using the built-in powers of Angular.

# The search component

As planned, we will have to separate the search feature from the Members component. To do that, let's create a new component called `SearchComponent` at `app/search/search.component.ts` and create the search component's template file. The template file will simply contain the search form.

The search component file will have to import and inject the `MembersService`, as that will be used to perform the search based on the search query. The component will have the search query and will request the service for search and get the search result.

The search component's code will look like this:

```
import { Component } from '@angular/core';

import { MembersService, Person } from '../services/members.service';

@Component({
  selector: 'app-search',
  moduleId: module.id,
  templateUrl: 'search.component.html'
})
export class SearchComponent {
  query: string;
  memberList: Array<Person> = [];

  constructor(public membersService: MembersService) {

  }

  search() {
    this.doSearch();
  }

  doSearch(): void {
    this.membersService.searchQuery(this.query)
    .then(results => {
      this.memberList = results;
    });
  }

}
```

The template of the search component will look like this:

```
<form>
  <input type="search" [(ngModel)]="query" name="query"
(keyup.enter)="search()">
  <button type="button" (click)="search()">Search</button>
</form>
```

As long as our application output doesn't break, we will have to bind the search component to the members list page, as it was before. So, we will have to append the search component to the template of the Members component. In that case, it will become the child component of the Members component.

The template of the Members component will look like this:

```
<h2>Members</h2>
<app-search></app-search>
<table *ngIf="memberList" id="searchList">
  ......
</table>
```

# Enabling sharing between components

Now we have two separate components, the search and Members components. The search component has been appended to the Members component, but search results are not available in the Members component.

Search and Members are separate components, and there is no bridge between them. Both have an isolated scope to contain their elements and variables.

To share data between components, we need to enable communication between them. As explained before, Angular events will come to our rescue for us to enable communication between the search and Members component. From the search component, we will need to use Angular custom events to communicate with its parent component, `MembersComponent`.

# Communicating with the parent component

The search component is the child component of the Members component. They need to communicate with each other to share data. We will need to use custom events with the help of the Angular `EventEmiiter` API so that we can emit the search result from the search component after getting the result. And besides that, we will need to use the `@OutPut` decorator to set the search result as the output to use for the parent component.

To use both, we will need to import both from Angular core. Then, we will need to set `@Output` searchResult to a new instance of EventEmitter. This `@Output` decorator makes the `searchResult` property available as an event binding.

When the search component updates the search result, we would like to tell the parent component that the `searchResult` event has happened. To do so, we will need to call `emit(data)` with the `searchResult` we have declared as the Emitter object with the Output decorator. The `emit()` method is used to notify every time the result has been passed via a custom event.

Now, the Members component can fetch the `$event` object, as we've passed it into the template using `(searchRessult)="anyMethod($event);`.

After updating with `EventEmitter`, the search component will look like this:

```
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
  ..................
})
export class SearchComponent {
  ..............
  @Output() searchResult: EventEmitter<any> = new EventEmitter();

  doSearch(): void {
    this.membersService.searchQuery(this.query)
    .then(results => {
      this.memberList = results;
      this.searchResult.emit(this.memberList));
    });
  }

}
```

Now it's time to communicate with the `MembersComponent`. Let's declare the `onSearch()` method in the member component, which will accept the event as an argument.

The Members component will change to the following:

```
export class MembersComponent implements OnInit {
  ngOnInit() {
    this.getMembers();
  }

  onSearch(searchResult) {
    this.memberList = searchResult;
  }

  getMembers() {
    this.membersService.getData()
    .then(data => {
      data.map(item => {
        this.memberList.push(item);
      });
    })
    return this.memberList;
  }

}
```

As we are appending the search component from the members template, let's hook the `onSearch` function to the search component tag. We will call this `(searchResult)`–with parentheses around it–to tell Angular that this is an event binding.

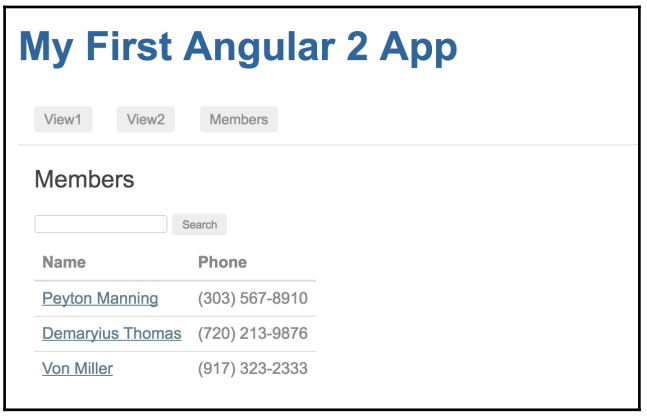
The template of the search component will look like this:

```
<h2>Members</h2>
<app-search (searchResult)="onSearch($event)" ></app-search>
<table *ngIf="memberList" id="searchList">
  ......
</table>
```

# Check output after refactoring

The search application will be rebranded into a store application instead of rewriting the search functionality that has already been written. In order to leverage the existing search project, it will be copied into a new project file. Then, the new project will use the tests to drive the development changes and refactoring. The refactoring steps have been left out, but a review of the code will show how the code and tests were modified to create the product application.

Time to run it and see how the service works with the Members component. Let's point our browser to `http://localhost:3000/members`.




We have the e2e test, which will confirm that everything's going well with the new changes:

```
$ npm run e2e
```

Yes, we can see that everything passes successfully:

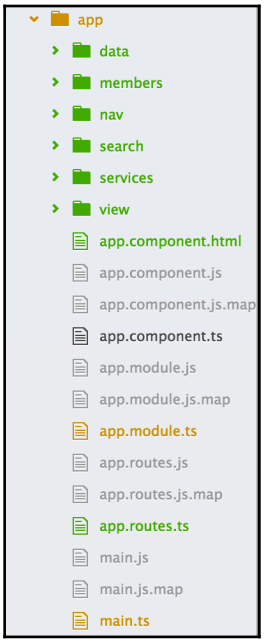
```
[1] [23:48:38] I/direct - Using ChromeDriver directly...
[1] [23:48:38] I/launcher - Running 1 instances of WebDriver
[1] AppDir: ./
[1] Started
[1]
[1]   Suite: Given should test the search feature
[1] .
[1]     passed - should have an input and search button
[1] .
[1]     passed - There should be one item in search result
[1]   Suite passed: Given should test the search feature
[1]   Suite: Given should test the search result in details view
[1] .
[1]     passed - should be load the person details page
[1]   Suite passed: Given should test the search result in details view
[1]   Suite: Given views should flip through navigation in
[1] .
[1]     passed - View1 should load and visible initially
[1] .    passed - View1 should have body content as expected
[1] .
[1]     passed - Should flipped to View2 and view2 should visible
[1] .
[1]     passed - Should flipped to View2 and should have body content as expected
[1] .
[1]     passed - Should flipped to View1 again and should visible
[1]   Suite passed: Given views should flip through navigation in
[1]
[1]
[1]
[1] 8 specs, 0 failures
[1] Finished in 17.085 seconds
[1]
[1]
[1] [23:49:00] I/launcher - 0 instance(s) of WebDriver still running
[1] [23:49:00] I/launcher - chrome #01 passed
```

Yes! Our code refactoring hasn't affected our expected behavior.

**Current project directory**

We have updated and refactored the code, for which we have some new components, services, and so on. Now, we will have a new project structure, separating the logic and decoupling the components.

Our current directory structure looks like this:




# Moving ahead

In this book, I have tried to cover the topics to a certain level so that anyone can start with test-driven development based on Angular. But there are a lot of things we have skipped, most importantly, rxJS.

rxJS is a separate module based on reactive programming. So, we will need to be familiar with reactive programming to understand it.

# Observables

HTTP requests by default return observables as responses in Angular instead of resolved promises. Since we didn't look at rxJS here, we skipped observables and converted the responses to promises. But we should learn how observables work with Angular.

# Publishing and subscribing

Publishing and subscribing messages is a powerful tool, but as with anything, when used the wrong way, it can lead to a mess.

There are two ways in which messages can be published: emit or broadcast. It is important to know the difference, as both work slightly differently and they may affect the performance of our application.

# Self-test questions

Q1. A callback function refers to a function that is called after an asynchronous function completes.

- True
- False

Q2. Asynchronous functions always complete in the order in which they were called.

- True
- False

Q3. There is a module called `MockBackend` to fake HTTP calls in Angular for unit testing.

- True
- False

Q4. In Angular, the `EventEmitter` API is used for component communication.

- True
- False

# Summary

In this chapter, we explored services in Angular and the power of events. We also saw some examples of the separation of code with services and events.

In addition, we looked at different types of testing for Angular components and wrote unit tests for the Angular router, and we integrated it with application components and navigation. We also explored further configuration of Karma in order to use its features.

Now that we're at the end of the book, it's time to apply our knowledge in the real world. Before leaving, let's have a quick recap of what we have learned. We learned about TDD, how TDD works with JavaScript context, and the available testing tools, techniques, and frameworks. We learned about Karma and Protractor with real Angular projects. And now we know how to write unit and e2e tests for Angular projects.

This book showed you the path to practicing TDD; now it's your job to keep learning, improve on this knowledge, and practice more with complex projects to be more confident with TDD.

# Index

## D



## E



## F



## G



## H



## I



## J

# K



# L



# M



# N



# O



# P

# Q



# R



# S

# T



# U



# W