AngularJS Test-driven Development

Starting with reviewing the test-driven development (TDD) life cycle, you will learn how Karma and Protractor make your life easier while running JavaScript unit tests. You will learn how Protractor is different from Selenium and how to test it entirely. This book is a walk-through to using TDD to build an AngularJS application containing a controller, model, and scope.

Building on the initial foundational aspects, you will expand to include testing for multiple controllers, partial views, location references, CSS, and the HTML element. In addition, you will also configure Karma file watching to automate testing and tackle components of AngularJS (controller, service, model, and broadcasting) using TDD. At the end of this book, you will extend explore how to pull data using an external API, setting up and configuring Protractor to use a standalone Selenium server, and setting up Travis CI and Karma to test your application.

This book is a complete guide to testing techniques using Karma for unit testing and performing end-to-end testing with Protractor.

Who this book is written for

This book is for developers who want to learn about AngularJS development by applying testing techniques. You are assumed to have a basic knowledge and understanding of HTML, JavaScript, and AngularJS.

What you will learn from this book

- Implement unit and end-to-end testing with Karma and Protractor
- Understand the workings of the file auto-watch feature and headless browser testing with Karma
- Develop AngularJS controllers, services, and other components using the TDD approach
- Implement the testing techniques with mocks, broadcast events, and asynchronous behavior
- Integrate REST-based services and APIs into an application to extract data
- Automate Karma unit tests with Travis CI
- Get to know about TDD techniques, the TDD life cycle, and its power through clear examples to enhance your AngularJS applications

Implement the best practices to improve your AngularJS applications using test-driven development

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 “Introduction to Test-driven Development”
- A synopsis of the book’s content
- More information on AngularJS Test-driven Development

About the Author

Tim Chaplin lives and breathes software solutions and innovations. During the day, he works with Fortune 100 enterprise applications, and in the evening, he perfects his craft by contributing to and distributing open source software, writing, and constantly looking for ways to increase his knowledge of technology and the world. At an early age, Tim began developing software and has been hooked on it since. Tim is an established conference speaker who has extensive experience in developing and leading AngularJS projects. He has a wide background of JavaScript, C#, Java, and C++ languages. Tim specializes in leading code quality and testing throughout all his applications. After attending California State University, Chico, he has gone on to work in Shanghai, Los Angeles, and London.

I would like to thank my wife, Pierra, for always making me think and dream bigger. I would also like to thank my family for their constant love and support. Pops, this one's for you babe.
AngularJS Test-driven Development

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

What This Book Covers

This book is basically split into two parts. The initial chapters focus on the TDD life cycle, and how Karma and Protractor fit into the life cycle and development of an AngularJS application. As we proceed, you'll get a step-by-step approach to AngularJS TDD using Karma and Protractor. Each of the chapters builds up on the previous one and introduces how to test several different AngularJS components.

Chapter 1, Introduction to Test-driven Development, is an introduction to the concepts of TDD and testing techniques.

Chapter 2, The Karma Way, explores the origins of Karma and why it is an essential tool for any AngularJS project.

Chapter 3, End-to-end Testing with Protractor, introduces the simplicity of Protractor, an end-to-end testing tool built specifically for AngularJS.

Chapter 4, The First Steps, covers the TDD journey and shows the fundamentals and tools in action.

Chapter 5, Flip Flop, expands to include testing for multiple controllers, partial views, location references, CSS, and HTML element building on the initial foundational aspects learned in the previous chapter.

Chapter 6, Telling the World, dives into communicating across controllers, and testing services and broadcasting.

Chapter 7, Give Me Some Data, dives into how to apply several of the concepts shown previously, and extend them to pull data using an external API.

Appendix A, Integrating Selenium Server with Protractor, walks through setting up and configuring Protractor to use a standalone Selenium server.

Appendix B, Automating Karma Unit Testing on Commit, covers how to set up Travis CI, a platform for continuous integration, and setting up Karma to test your application.
Introduction to Test-driven Development

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

This chapter introduces you to the fundamentals of test-driven development with AngularJS including:

- An overview of test-driven development (TDD)
- The TDD life cycle: test first, make it run, make it better
- Common testing techniques

An overview of TDD

TDD is not used only to develop software. The fundamental principles can be seen in many industries. This section will explore the fundamentals of TDD and how they are applied by a tailor.

Fundamentals of TDD

Know what to code before you code. This may sound cliché, 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 have been met.
Here are a couple of clear benefits of using TDD:

- **Knowing before you code**: A test provides a clear vision of what code needs to do in order to be successful. Setting up tests first allows focus on only components that have been defined in 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.

- **Documentation**: Tests define expectations that a particular object or function must meet. The 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 success

TDD is not just a software development practice. The 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 a tailor takes to make a suit:

1. **Test 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**:
   - Measuring the fabric and cut
   - Selecting the fabric based on the desired style
   - Measuring the fabric based on the customer's waist and legs
   - Cutting the fabric based on the measurements
3. **Refactoring**:
   - Comparing the resulting product to the expected style, reviewing, and making changes
   - 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 pants
   - **Making 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 up the raw material. Imagine for a moment if 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? Measure twice, cut once.

Software developers can choose from an endless amount of approaches to use before starting developing. 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 than the specification. With a TDD approach (test first, make it run, and make it better), 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. Develop like a professional and follow the practices that will help you write quality software.
Diving in
It is time to dive into some actual code. This walk-through will take you through adding the multiplication functionality to a calculator. Remember the TDD lifecycle: test first, make it run, and make it better.

Setting up the test
The initial calculator is in a file called calculator.js and is initialized as an object as follows:

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

The test will be run through a web browser using a basic HTML page. Create a web page and import calculator.js to test it. Save the web page as testRunner.html. To run the test, open a browser and run testRunner.html. Here is the code for testRunner.html:

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

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

Now that the project is set up, the next step is to create the development to-do list.

Creating a development to-do list
A development to-do list helps organize and focus your tasks. It also provides a place to write down ideas during the development process.

Here is the initial step for creating a development to-do list:

- Add multiplication functionality: $3 \times 3 = 9$

The preceding list describes what needs to be done. It also provides a clear example of how to verify multiplication: $3 \times 3 = 9$. 
Test first
Although you can write the multiplication function quickly, remember that once the habit of TDD is set in place, it will be just as quick to write the test and code. Here are the steps for the first test:

1. Open calculator.js.
2. Create a new function to test multiplying 3 * 3:
   ```javascript
   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');
     }
   }
   ```

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. Remember, 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 through 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 will be focused on making the test run.

Making it run
This step is about making the test run, just as the tailor did with the suit. The measurements were taken during the test step, and now the application can be molded to fit the measurements. Here are the steps to run the test:

1. Open the browser with testRunner.html.
2. Open the JavaScript developer Console window.
Introduction to Test-driven Development

The test throws an error, as shown in the following screenshot:

![Screenshot of test error]

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 smallest change to get a test to pass. There is no need to actually implement the multiplication logic. This may seem unintuitive. The point is 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 it meets the expectations.

What is the smallest 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` function, it will confirm the application wiring. In addition, after you have passed the test, making future changes will be easy as you have to simply keep the test passing!

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

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

In the browser, the JavaScript console reruns the test. The result should be as follows:

![Screenshot of test result]

Yes! The test passed. Time to cross out the first item from the to-do list:

- Add multiplication functionality: $3 \times 3 = 9$

Now that there is a passing test, the next step will be to remove the hardcoded value in the `multiply` function.
Making it better

The refactoring step needs to remove the hardcoded return value of the multiply function. The required logic is as follows:

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

Rerun the tests and confirm the test passes. Excellent! Now the multiply function is complete. Here is the full code for the calculator and test:

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

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

multipleTest1();
```

Testing techniques

It is important to understand some fundamental techniques and approaches to testing. This section will walk you through a couple of examples of techniques that will be leveraged in this book. This includes:

- Testing doubles with Jasmine spies
- Refactoring
- Building patterns
In addition, here are additional terms that will be used:

- **Function under test**: This is the function being tested. It is also referred to as system under test, object under test, and so on.

- **The 3 A's (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 A's will be discussed further in Chapter 2, *The Karma Way*.

**Testing with a framework**

Although a simple web page can be used to perform tests, as seen earlier in this chapter, it is much 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 more. This book uses Jasmine as the test framework. Jasmine is a behavior-driven testing framework. It is highly compatible with testing AngularJS applications. In Chapter 2, *The Karma Way*, we will take a more in-depth look at Jasmine.

**Testing doubles with Jasmine spies**

A test double is an object that acts and is used in place of another object. Take a look at the following object that needs to be tested:

```javascript
var objectUnderTest = {
    someFunction : function(){}
};
```

Using a test double, you can determine the number of times `someFunction` gets called. Here is an example:

```javascript
var objectUnderTest = {
    someFunction : function(){}
};

jasmine.spyOn(objectUnderTest,'someFunction');
objectUnderTest.someFunction();
objectUnderTest.someFunction();

console.log(objectUnderTest.someFunction.count);
```
The preceding code creates a test double using a Jasmine spy (jasmine.spyOn). The test double is then used to determine the number of times someFunction gets called. A Jasmine test double offers the following features and more:

- 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, you 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, you can specify exactly what a method should return for a given test. Here is an example function:

```javascript
var objectUnderTest = {
    someFunction : function(){ return 'stub me!'; } 
};
```

The preceding object (objectUnderTest) has a function (someFunction) that needs to be stubbed. Here is how you can stub the return value using Jasmine:

```javascript
jasmine.spyOn(objectUnderTest,'someFunction')
 .and
 .returnValue('stubbed value');
```

Now, when objectUnderTest.someFunction is called, stubbed value will be returned. Here is how the preceding stubbed value can be confirmed using console.log:

```javascript
var objectUnderTest = {
    someFunction : function(){ return 'stub me!'; } 
};

//before the return value is stubbed
console.log(objectUnderTest.someFunction());  //displays 'stub me'

jasmine.spyOn(objectUnderTest,'someFunction')
 .and
 .returnValue('stubbed value');

//After the return value is stubbed
console.log(objectUnderTest.someFunction());  //displays '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:

```javascript
var objectUnderTest = {
    someFunction : function(arg1,arg2){}
};
```

Here are the steps to test the arguments the preceding function is called with:

1. Create a spy so that the arguments called can be captured:
   ```javascript
   jasmine.spyOn(objectUnderTest,'someFunction');
   ```

2. Then to access the arguments, do the following:
   ```javascript
   //Get the arguments for the first call of the function
   var callArgs = objectUnderTest.someFunction.call.argsFor(0);

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

3. Here is how the arguments can be displayed using `console.log`:
   ```javascript
   var objectUnderTest = {
       someFunction : function(arg1,arg2){}
   };

   //create the spy
   jasmine.spyOn(objectUnderTest,'someFunction');

   //Call the method with specific arguments
   objectUnderTest.someFunction('param1','param2');

   //Get the arguments for the first call of the function
   var callArgs = objectUnderTest.someFunction.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 aesthetic of a piece of code. The TDD life cycle step of "making it better" is primarily concerned with refactoring. This section will walk you through a refactoring example. Here is an example of a function that needs to be refactored:

```javascript
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 the 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:

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

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

3. Remove unnecessary complexity. In this case, the if conditional statement can be removed completely:

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

4. Reflect on the result.

   At this point, the refactor is complete, and the function's purpose should jump out at you. The remaining 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 issues in code and how to improve them. Other examples will be used throughout this book.

**Building with a builder**

The builder pattern uses a builder object to create another object. Imagine an object with ten 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 builder object in the validate method:

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

The book object has three properties: id, author, and dateTime. From a testing perspective, you would want the ability to create a valid object, that is, one that has all the fields defined. You may also want to create an invalid object with missing properties, or you may want to set certain values in the object to test the validation logic, that is, dateTime is an actual date.

Here are the steps to create a builder for the dateTime object:

1. Create a builder function:
   ```javascript
   var bookBuilder = function();
   ```

2. Create a valid object within the builder:
   ```javascript
   var bookBuilder = function(){
      var _resultBook = {
         id: 1,
         author: 'Any Author',
         dateTime: new Date()
      };
   }
   ```
3. Create a function to return the built object:

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

4. Create another function to set the `_resultBook` author field:

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

5. Make the function fluent so that calls can be chained:

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

6. A setter function will also be created for `dateTime`:

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

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

```javascript
var builtBook = bookBuilder.setAuthor('Tim Chaplin')
  .setDateTime(new Date())
  .build();
```
Introduction to Test-driven Development

The preceding builder can now be used throughout your tests to create a single consistent object. Here is the complete builder for your reference:

```javascript
var bookBuilder = function(){
    var _resultBook = {
        id: 1,
        author: 'Any Author',
        dateTime: new DateTime()
    };

    this.build = function(){
        return _resultBook;
    };

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

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

**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 you have purchased. 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.

**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 A's 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, make it better) and showed how the same steps are used by a tailor. Finally, it looked over some of the testing techniques that will be discussed throughout this book including:

- Test doubles
- Refactoring
- Building patterns

Although TDD is a huge topic, this book is solely focused on the TDD principles and practices to be used with AngularJS. In the next chapter, you will start the journey and see how to set up the Karma test runner.
Where to buy this book

You can buy AngularJS Test-driven Development from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.