**1 Introduction to testing Angular applications**

**https://jasmine.github.io/**

**https://livebook.manning.com/book/testing-angular-applications**

**https://www.manning.com/books/testing-angular-applications**

**>ng test**

**>ng e2e**

**Karma.conf.js –file**

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

Poorly written code, buggy functionality, and bad refactoring practices can lead to unreliable applications. Writing good tests will help detect these types of problems and prevent them from negatively affecting your application. It’s vital that you thoroughly test your application if you want to make it sustainable and supportable for years to come. A core purpose of writing tests is to help guard against breaking application functionality when you have to add new features or make bug fixes later on.

If you’ve developed an Angular application, you may know that Angular is a great framework for building testable web and mobile web applications. One of the goals in writing Angular was to make it a testable framework, and it shows.

Although testing Angular applications is of utmost importance, figuring out how to do that has been challenging until now. You may have been able to find a blog post or two, perhaps a video, but generally materials have been lacking to help guide you through all the different aspects of testing in one place. Well, you’re in luck! In your hands (or on your screen), you hold the key to getting started with testing Angular applications.

This book will help you build a foundation for testing the most important parts of Angular applications with confidence. We assume that you have some familiarity with the Angular framework, TypeScript, and command-line tools. If you haven’t written a test, this book will teach you enough fundamentals to get you started.

If you don’t have experience with Angular, now is a great time to learn about the Angular applications. For newbies, we would encourage you to walk through the tutorials and introductory information you can find at [https://angular.io](https://angular.io/).

In this first chapter, you’ll get an overview of testing Angular applications, take a brief look at TypeScript, learn about the testing tools you’ll use, and be introduced to unit and end-to-end (E2E) tests. Let's get started!

**1.1 Angular testing overview**

Most Angular testing you’ll find out there involves two types of tests: unit tests and E2E tests. The bulk of this book will revolve around those two types.

This book is separated into two parts. The first part covers unit testing, which tests units of code. You’ll learn how to create unit tests for components, directives, pipes, services, and routing—using testing tools like Karma and Jasmine—and run those tests using the Angular command-line interface (CLI). The following list breaks down each of the testable concepts we’ll cover in part 1 of the book:

* *Components* —Chunks of code that you can use to encapsulate certain functionality that you can then reuse throughout the application. Components are types of directives (see next bullet), except they include a view or HTML template.
* *Directives* —Used to manipulate elements that exist in the DOM or can add elements to or remove them from the DOM. Examples of directives included with Angular are ngFor, ngIf, and ngShow.
* *Pipes* —Used to transform data. For example, say you want to turn an integer into currency. You would use a currency filter pipe to turn 15 into $15.00.
* *Services* —Although services technically don’t exist in Angular, the concept is still important. You’ll use services to fetch data and then inject it into your components.
* *Routing* —Allows users to navigate from one view to the next as they perform tasks in the web application.

In the second part of the book, we’ll dive into E2E testing using the Protractor framework. You’ll get practice writing tests that behave as if the interactions were coming from the user in a browser.

As for which version of Angular you’ll be using, this book is written to be compatible with versions of Angular 2 and later. Angular 2 was a complete rewrite from AngularJS 1.x, so that’s the base for the current version.

**NOTE**

It's just *Angular* now. In the past, people have referred to Angular as AngularJS, Angular 1, Angular 2, Angular 4, and so on. From here on out, we’ll use AngularJS when we mean Angular 1.x, and Angular when we’re talking about versions 2 and higher. To read more about the decision, check out this blog post: <http://angularjs.blogspot.com/2016/12/ok-let-me-explain-its-going-to-be.html>.

In the next section, we’ll look at TypeScript, which is the language you’ll use to write tests in this book.

**1.2 Getting friendly with TypeScript**

TypeScript is an open source language created at Microsoft in 2012 by Anders Hejlsberg, who also created C#.[1](https://livebook.manning.com/book/testing-angular-applications/chapter-1/c01-footnote-1)  The major problem that Hejlsberg attempted to solve with TypeScript is that JavaScript was never meant to be used with large-scale applications. The first version of JavaScript was created in 1995 in 10 days by Brendan Eich[2](https://livebook.manning.com/book/testing-angular-applications/chapter-1/c01-footnote-2)  and was meant to be used as a scripting language for adding interactivity to web pages.

[1](https://livebook.manning.com/book/testing-angular-applications/chapter-1/c01-footnoteref-1)  See “TypeScript,” Wikipedia, <https://en.wikipedia.org/wiki/TypeScript>.

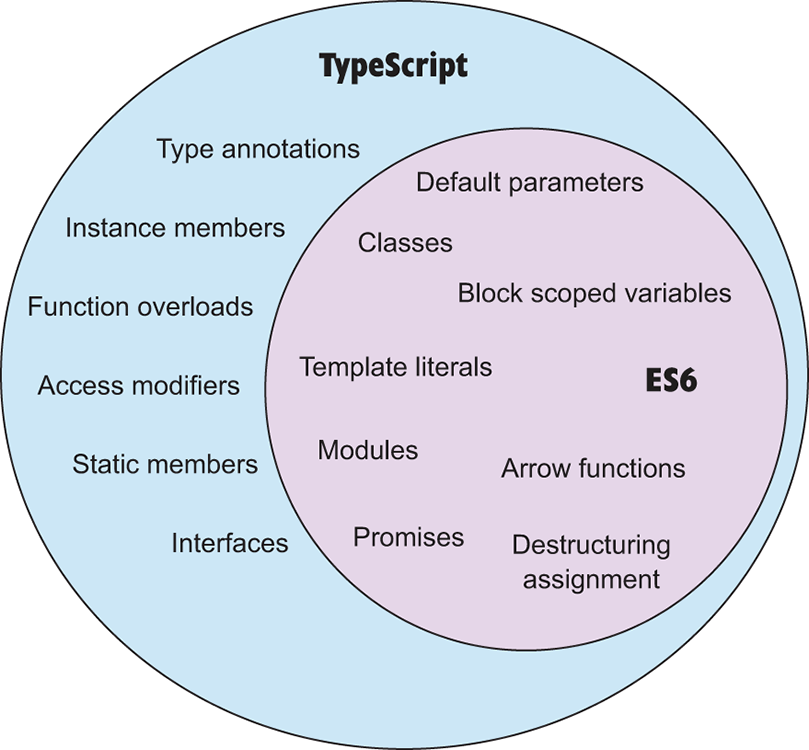
[2](https://livebook.manning.com/book/testing-angular-applications/chapter-1/c01-footnoteref-2)  See “JavaScript,” Wikipedia,[https://en.wikipedia.org/wiki/JavaScript#Beginnings\_at\_Netscape](https://en.wikipedia.org/wiki/JavaScript" \l "Beginnings_at_Netscape).

Although you can build Angular applications with native JavaScript, we recommend you use TypeScript, because most of the examples, tutorials, documentation, code examples, and so on in the book will use TypeScript. In addition, the Angular framework itself is built with TypeScript.

TypeScript adds benefits needed for enterprise applications, such as annotations, static typing, and classical object-oriented features like interfaces and code encapsulation, while still providing the key features of JavaScript.

You’ll find the syntax of TypeScript to be much like that of JavaScript, because TypeScript is a superset of JavaScript. [Figure 1.1](https://livebook.manning.com/book/testing-angular-applications/chapter-1/figure1.1) shows how TypeScript’s key features (outer circle) encompass the key features of the ES6 version of JavaScript (inner circle).

[Figure 1.1](https://livebook.manning.com/book/testing-angular-applications/chapter-1/#figureanchor1.1) TypeScript is a superset of JavaScript version ES6, so they share much of the same syntax.



TypeScript includes most of the key ES6 features, so almost any valid ES6 or ES5 feature is also valid TypeScript. You could have a perfectly valid TypeScript file that uses only standard JavaScript syntax.

TypeScript compiles to JavaScript, and the Angular CLI will compile TypeScript automatically out of the box. If you’re familiar with JavaScript and have some knowledge of object-oriented programming (OOP), you should be able to pick up TypeScript quickly. If you’re new to OOP, that’s fine. It may just take a little more work for you to get up to speed.

The following listing creates a simple class called Cat that demonstrates what a class looks like in TypeScript.

**Listing 1.1 Simple TypeScript class example**

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

**export** **class** **Cat** {

private \_name: string = '';    #1    #2

constructor(name? : string) {    #1

**this**.\_name = name;

}

get name() : string {

**return** **this**.\_name;

}

set name(name : string) {

**this**.\_name = name;

}

toString() : string {

**return** `This cat's name is ${this.\_name}!`;

}

}

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

**1**

**3**

**4**

[copy](javascript:void(0))

This example demonstrates several features. The class, get, set, and constructor keywords, along with the template string found in the toString() method, are all part of the ES6 version of JavaScript, so those keywords aren’t exclusive to TypeScript.

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

In the Cat class, notice that the keyword string is sprinkled throughout the example. This is how you specify the expected types in TypeScript. You can specify return types, variable types, and parameter types within your code. The ability to specify types—type annotations—is the key advantage of using TypeScript, hence the name.

We’ll be covering TypeScript throughout the book, but if you want to get your feet wet with TypeScript outside of that, visit [http://www.typescriptlang.org](http://www.typescriptlang.org/). The website offers a Playground feature where you can write and test code.

Now that you’ve learned a bit about TypeScript, let’s look at the different types of tests that you’ll be writing in this book.

**1.3 A closer look at test types**

Unit tests and E2E tests are the two types of tests you’ll see in the wild for testing Angular applications. In this section, we’ll look a little more closely at those two types.

**1.3.1 Unit tests**

You write unit tests to test the functionality of basic parts or units of code. Each unit test should only test one responsibility of the source code. You can test functions, methods, objects, types, values, and more with unit tests. The advantages of using unit tests are that they tend to be fast, reliable, and repeatable, if you write them correctly and run them in the right environment.

Jasmine, a behavior-driven development framework for testing JavaScript code, is the framework you’ll use in this book for writing unit tests. The following listing shows the code for a basic unit test written using the Jasmine framework.

**Listing 1.2 Example of a simple unit test**

1

2

3

4

5

describe('super basic test', () => {

it('true is true', () => {

expect(true).toEqual(false);

});

});

**1**

[copy](javascript:void(0))

All you’re doing in [listing 1.2](https://livebook.manning.com/book/testing-angular-applications/chapter-1/listing1.2) is checking to see that the Boolean value true is equal to the Boolean value true. This test serves as a sanity check and nothing more. You can use a *sanity check* to see if all the parts of your testing environment are set up correctly, and you use it when you’re only attempting to get a basic test to pass. You wouldn’t want to add a test this simple to a production application.

The next listing shows a slightly more sophisticated unit test that tests getters and setters for the name instance found in the Cat class you created in [listing 1.1](https://livebook.manning.com/book/testing-angular-applications/chapter-1/listing1.1).

**Listing 1.3 Better example of a unit test**

1

2

3

4

5

6

7

8

9

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

describe('Test Cat getters and setters.', () => {

it('The cat name should be Gracie', () => {

const cat = **new** Cat();

cat.name = 'Gracie';

expect(cat.name).toEqual('Gracie');

});

});

**1**

[copy](javascript:void(0))

Although unit tests tend to be reliable, they aren’t the best type of test for reproducing real user interactions.

**1.3.2 E2E tests**

You use E2E tests to test the functionality of an application by simulating the behavior of an end user. For example, you might have an E2E test check if a modal correctly appears after a form is submitted or a page renders certain elements on page load, such as buttons or text.

E2E tests do a good job with testing applications from an end user’s standpoint, but they can run slowly, and that slowness can be the source of false positives that fail tests because of timeout issues. E2E tests’ timing issues make it preferable to write unit tests instead of E2E tests whenever possible.

For writing the E2E tests in this book, you’ll use the Protractor E2E test framework developed by the Angular team.

The following listing shows a sample E2E test written using Protractor that checks the sample project’s website and makes sure the title of the page is equal to Contacts App Starter.

**Listing 1.4 Sample E2E test using Protractor**

1

2

3

4

5

6

7

8

9

10

11

12

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

describe('Contacts App title test', () => {

it('Title should be correct', () => {

const appUrl = 'https://testing-angular-applications.github.io';

const expectedTitle = 'Contacts App Starter';

browser.get(appUrl);

browser.getTitle().**then**((actualTitle) => {

expect(actualTitle).toEqual(expectedTitle);

});

});

});

**1**

**2**

**3**

[copy](javascript:void(0))

We’ll explore unit tests and E2E tests in much greater detail throughout the book. If you understand these two types of tests, you’ll understand much of Angular testing.

**1.3.3 Unit tests vs. E2E tests**

Unit tests and E2E tests have different advantages. We’ve discussed these advantages a little already, but in [table 1.2](https://livebook.manning.com/book/testing-angular-applications/chapter-1/table1.2), we summarize the pros and cons of the different types of tests.

**Table 1.2: Advantages to using unit tests vs. E2E tests**

| **Feature** | **Unit tests** | **E2E tests** |
| --- | --- | --- |
| Speed | Tend to be faster than E2E tests. | Tend to be slower than unit tests. |
| Reliability | Tend to be more reliable than E2E tests. | Tests can be flaky and fail because they may time out while executing. |
| Helping enforce code quality | Writing tests can help identify needlessly complex code that may be difficult to test. | Tests from the browser won’t help write better code because you’re testing the app as a whole from the outside. |
| Cost-effectiveness | More cost-effective because of developer time to write tests, execution of tests, and reliability. | Less cost-effective because it takes longer to write tests, execution of tests is slow, and tests can be flaky. |
| Mimicking user interactions | Tests can mimic user interactions but can be hard to use to check complex interactions. | Mimicking user interactions is E2E tests’ forte and what they’re made for. |

Let’s discuss each of these features one by one:

* *Speed* —Because unit tests operate on small chunks of code, they can run quickly. E2E tests rely on testing through a browser, so they tend to be slower.
* *Reliability* —Because E2E tests tend to involve more dependencies and complex interactions, they can be flaky, which can lead to false positives. Running unit tests rarely results in false positives. If a well-written unit test fails, you can trust that there’s a problem with the code.
* *Helping enforce code quality* —One of the main benefits of writing tests is that it helps enforce code quality. Writing unit tests can help identify needlessly complex code that may be difficult to test. As a general rule, if you’re finding it hard to write unit tests, your code may be too complex and may need to be refactored.

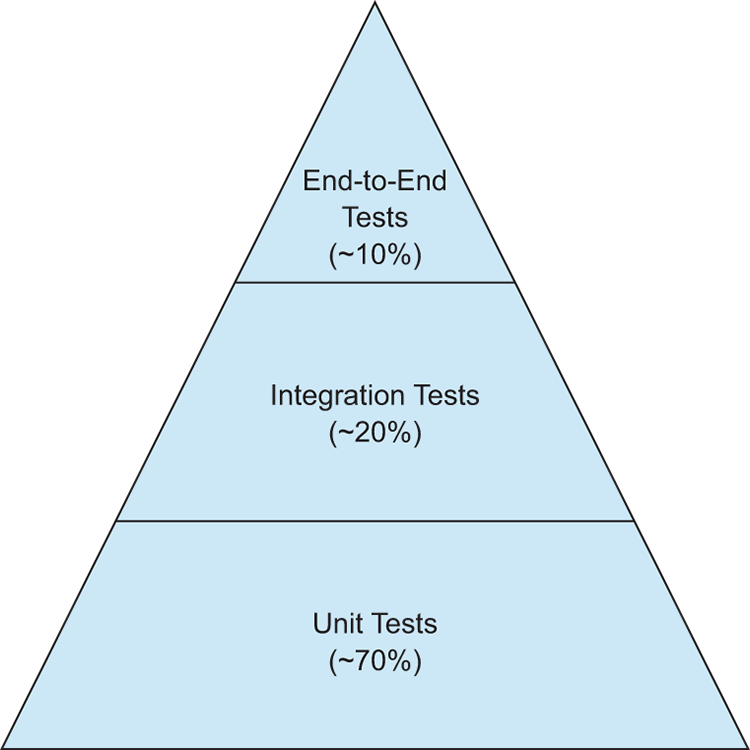
Writing E2E tests won’t help you write better quality code per se. Because E2E tests test from the browser standpoint, they don’t directly test your code.

* *Cost-effectiveness* —Because E2E tests take longer to run and can fail at random times, a cost is associated with that time. It also can take longer to write such tests, because they may build on other complex interactions that can fail, so development costs also can be higher when it comes to writing E2E tests.
* *Mimicking user interactions* —Mimicking user interactions with the UI is where E2E tests shine. Using Protractor, you can write and run tests as if a real user were interacting with the user interface. You can simulate user interactions using unit tests, but it’ll likely be easier to write E2E tests for that purpose because that’s what they’re made for.

Both types of tests are important to have to thoroughly test your applications. You can test a lot of the functionality that a user would perform by writing unit tests, but you should test key functionality with E2E tests.

Generally, you want to have more unit tests than E2E tests in your project. A software developer named Mike Cohn created a testing pyramid to show how the different types of tests should be broken down. In [figure 1.2](https://livebook.manning.com/book/testing-angular-applications/chapter-1/figure1.2), you can see approximate percentages for the different tests you should include in your project.

[Figure 1.2](https://livebook.manning.com/book/testing-angular-applications/chapter-1/#figureanchor1.2) Testing pyramid of the recommended distribution for the different types of tests in your project



You may have noticed that the pyramid includes integration tests. Integration tests are used to test that a whole system works correctly. For our purposes, we’ll roll the integration tests into E2E tests, because E2E tests test the entire UI system.

That gives you a basic background on Angular testing. In the next chapter, you’ll get the chance to dive in to writing your first real tests.

**Summary**

* Angular provides a great framework for building testable web and mobile applications. It relies heavily on the concept of components for developing applications.
* You’ll run across two types of tests when developing Angular applications: unit tests and E2E tests. You use unit tests to test the actual code, whereas you write E2E tests in a way that enables them to simulate user interactions.
* TypeScript is a language created by Microsoft that’s a superset of JavaScript. Angular is built with TypeScript, and although you can use native versions of JavaScript (ES5 or ES6) or compile-to-JavaScript languages like
* CoffeeScript or Dart, we recommend you use TypeScript to write your code. Most of the coding examples, tutorials, and blog posts for this book will use TypeScript, so it’s in your best interest to jump in and start using it.
* TypeScript comes with some nifty features not found in native JavaScript, such as the ability to assign types, declare private variables, and use object-oriented features, like interfaces.
* Protractor, Jasmine, and Karma are the primary tools that make the testing of Angular applications easy. These tools are written in JavaScript, and you’ll configure them to run automatically while you write your tests and during your build process.

### Test file name and location

Look inside the src/app folder.

The CLI generated a test file for the AppComponent named  app.component.spec.ts.

The test file extension **must be**.spec.ts so that tooling can identify it as a file with tests (AKA, a spec file).

The app.component.ts and app.component.spec.ts files are siblings in the same folder. The root file names (app.component) are the same for both files.

Adopt these two conventions in your own projects for every kind of test file.

#### Place your spec file next to the file it tests

It's a good idea to put unit test spec files in the same folder as the application source code files that they test:

* Such tests are easy to find.
* You see at a glance if a part of your application lacks tests.
* Nearby tests can reveal how a part works in context.
* When you move the source (inevitable), you remember to move the test.
* When you rename the source file (inevitable), you remember to rename the test file.

#### Place your spec files in a test folder

Application integration specs can test the interactions of multiple parts spread across folders and modules. They don't really belong to any part in particular, so they don't have a natural home next to any one file.

It's often better to create an appropriate folder for them in the tests directory.

Of course specs that test the test helpers belong in the test folder, next to their corresponding helper files.

Code Coverage:

# Find out how much code you're testing

The CLI can run unit tests and create code coverage reports. Code coverage reports show you any parts of your code base that may not be properly tested by your unit tests.

For the sample app that the testing guides describe, see the [sample app](https://angular.io/generated/live-examples/testing/stackblitz.html).

For the tests features in the testing guides, see [tests](https://angular.io/generated/live-examples/testing/specs.stackblitz.html).

To generate a coverage report run the following command in the root of your project.

content\_copyng test --no-watch --code-coverage

When the tests are complete, the command creates a new /coverage folder in the project. Open the index.html file to see a report with your source code and code coverage values.

If you want to create code-coverage reports every time you test, you can set the following option in the CLI configuration file, angular.json:

content\_copy"test": {

"options": {

"codeCoverage": true

}

}

## Code coverage enforcement

The code coverage percentages let you estimate how much of your code is tested. If your team decides on a set minimum amount to be unit tested, you can enforce this minimum with the Angular CLI.

For example, suppose you want the code base to have a minimum of 80% code coverage. To enable this, open the [Karma](https://karma-runner.github.io/) test platform configuration file, karma.conf.js, and add the check property in the coverageReporter: key.

content\_copycoverageReporter: {

dir: require('path').join(\_\_dirname, './coverage/<project-name>'),

subdir: '.',

reporters: [

{ type: 'html' },

{ type: 'text-summary' }

],

check: {

[global](https://angular.io/api/core/global): {

statements: 80,

branches: 80,

functions: 80,

lines: 80

}

}

}

The check property causes the tool to enforce a minimum of 80% code coverage when the unit tests are run in the project.