

ANGULAR TESTING

SUCCINCTLY

BY **JOSEPH D. BOOTH**

Angular Testing Succinctly

By

Joseph D. Booth

Foreword by Daniel Jebaraj



Copyright © 2020 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.
ISBN: 978-1-64200-206-5

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.

Proofreader: Graham High, senior content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the <i>Succinctly</i> Series of Books.....	7
About the Author	9
Chapter 1 Introduction.....	10
Lint checking	10
Unit testing	10
End-to-end (E2E) testing	10
Setting up your environment.....	11
Exploring a project setup	11
Summary.....	13
Chapter 2 Lint Checking	14
Installation	15
Configuration	16
Rules.....	17
Fixers	21
Summary.....	23
Chapter 3 Other Linters	24
SonarLint.....	24
ESLint	26
Summary.....	27
Chapter 4 Test-Driven Development.....	28
ATM testing	28
Behavior-driven development.....	30
Summary.....	31
Chapter 5 Angular CLI	32

New project	32
Create a service	33
Create a component.....	33
Summary.....	34
Chapter 6 Sample Application	35
Main screen.....	35
Application files	36
Service	36
Components.....	38
What should we test?	38
Summary.....	39
Chapter 7 Unit Testing.....	40
Spec files.....	40
TestBed.....	41
Jasmine.....	43
Karma test runner.....	50
Code coverage	52
Designing your unit tests	54
Summary.....	55
Chapter 8 E2E Testing with Jasmine	56
Configuration	56
Test files.....	59
Protractor API.....	60
Summary.....	72
Chapter 9 E2E Testing with Cucumber.....	73
Feature files	74

Example feature file.....	78
Structure.....	78
Page object files	79
Installation	83
Configuration.....	83
Summary.....	85
Chapter 10 Planning E2E.....	86
What testing framework?	86
Base functionality	86
Templates	87
Summary.....	87
Chapter 11 Summary	88
TSLint.....	88
ESLint	88
SonarLint.....	88
Karma	89
Jasmine.....	89
Protractor	89
Cucumber.....	89
Summary.....	89
Appendix A Jasmine Syntax	90
Appendix B Jasmine Matchers	92
Appendix C Gherkin Syntax	94

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Joseph D. Booth has been programming since 1981 in a variety of languages including BASIC, Clipper, FoxPro, Delphi, Classic ASP, Visual Basic, JavaScript, Visual C#, and the .NET Framework. He has also worked in various database platforms, including DBASE, Paradox, Oracle, and SQL Server.

He is the author of [GitHub Succinctly](#), [Accounting Succinctly](#), [Regular Expressions Succinctly](#), [Visual Studio Add-Ins Succinctly](#), [Natural Language Processing Succinctly](#), [Angular Succinctly](#), and [SQL Server Metadata Succinctly](#) from Syncfusion, as well as six books on Clipper and FoxPro programming, network programming, and client/server development with Delphi. He has also written several third-party developer tools, including CLIPWKS, which allows developers to programmatically create and read native Lotus and Excel spreadsheet files from Clipper applications.

Joe has worked for a number of companies, including Yprime, Sperry Univac, MCI-WorldCom, Ronin, Harris Interactive, Thomas Jefferson University, People Metrics, and Investor Force. He is one of the primary authors of Results for Research (market-research software), PEPSys (industrial-distribution software), and a key contributor to AccuBuild (accounting software for the construction industry).

He has a background in accounting ([Accounting Succinctly](#)), having worked as a controller for several years in the industrial distribution field, but his real passion is computer programming.

In his spare time, Joe is an avid tennis player. He also practices yoga and martial arts, and plays with his grandkids, Blaire and Kaia. You can visit his [website](#) for more information.

Chapter 1 Introduction

Angular is a powerful framework for developing component-based web applications. In this book, we are going to focus on the tools and options for improving the quality of your Angular code base. There are several features that can be utilized to produce higher-quality, more-maintainable code.

Lint checking

Lint checking is a static analysis process that reviews your source code files to look for things like patterns and style issues that are present in your source. None of the code in your source files is executed; a set of patterns and rules is simply checked against the source code files. This type of checking won't detect runtime bugs per se, but will identify areas of the code that might cause unexpected behaviors.

Most lint checkers operate using a set of rules and allow you to customize the rules to your (and your team's) programming preferences. You can install lint checkers into most code editors to lint check during coding, or you can run lint checking as a separate process after saving your code.

Unit testing

Unit testing creates an instance of your component or service and runs the created instance, checking that your code works as expected. For example, if a component validates birth dates, you could have unit tests to ensure a valid date is provided, and that the date is in the past. Your tests should generally test both good, expected behavior and error conditions. The Angular CLI will generate a basic unit test file when you create a new project and/or new components and services.

Some advantages of unit testing are that errors are found earlier in the development cycle, and unit tests can be used for regression testing (confirming new code doesn't break previously working code). However, thorough unit testing can take significant time to write. Each Boolean condition would require two tests (one for the **IF** and another for the **ELSE** condition).

End-to-end (E2E) testing

End-to-end (E2E) testing integrates your components and the browser to provide an automated set of tests mimicking how a user will interact with your system. For example, an E2E test on a login process might open the browser, try to log in with invalid credentials, and confirm that an error message is displayed. Another test within the E2E process would confirm that logging in with valid credentials brings up the application splash screen or main panel.

Setting up your environment

Each process has a place in your development cycle and should be standard operating practice for Angular development.

Lint checking should always be performed first; it is quick and identifies potential code and style issues. The process is easily customizable, so you can define coding styles and standards that all members on the development team can adhere to.

Unit testing is generally quick to execute, ensuring the methods in your components do what they are expected to do. Unit tests are generally small and run quickly, and should be performed as each component is written or updated. However, unit tests test components in isolation, and don't check how the components will interact with each other.

End-to-end testing can be very slow, basically running the integrated application within a browser to ensure the components interact with each other and the application UI performs as expected.

While end-to-end testing is a good final, automated test feature, it still does not replace the need for manual, human testing.

Exploring a project setup

If you've used the Angular CLI to create a basic project, you will find that the configuration and setup work for lint checking, unit testing, and end-to-end testing is started for you.

tslint.json

This file will be found in the project root and contains a basic set of rules (using presets) and some overwritten rules to tweak the code for Angular modules. Within the **node_modules** folder, there is a directory called **tslint**. This folder contains the actual tslint program and library.

src/app

For each component created by the CLI, a .spec file with the same base component file name will be generated. You will typically see four files per component: a style sheet, the HTML template code, the component, and the specification file. Code Listing 1 shows a simple component.

Code Listing 1: Basic Angular component

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

@Component({
  selector: 'app-root',
```

```

    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
  })
  export class AppComponent {
    title = 'TestProject';
  }

```

The matching .spec file is shown in Listing 2, and we will cover the file in detail in Chapter 4. You can see the `it()` method with a text string to indicate what is being tested. These are very simple, basic tests, and you'll need to keep the file up to date as you build out your component.

Code Listing 2: Basic Angular test component

```

import { Component } from '@angular/core';
import { TestBed, async } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [ RouterTestingModule ],
      declarations: [ AppComponent ],
    }).compileComponents();
  }));

  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  });

  it(`should have as title 'TestProject'`, () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('TestProject');
  });

  it('should render title', () => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.querySelector('.content span').textContent)
      .toContain('TestProject app is running!');
  });

```

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

In general, the method provides a description of the test and a function to perform the actual test. The **expect** function indicates the expected result if the test is successful.

E2E folder

The E2E folder contains two files: the **protractor.config.js** file, which configures the Protractor tool to run the tests; and a **tsconfig.json** file, which extends the base `tsConfig` and customizes the compiler options necessary for running the E2E tests.

E2E\Src

The **src** folder within the E2E directory contains two files that are used to drive the E2E test process. The **app.po.ts** file contains a class representing the application and a few methods to perform operations within the browser. The **app.e2e-specs.ts** file is the actual TypeScript code that imports the application class from the **app.po.ts** file, and some classes from Protractor.

The syntax in the E2E file looks at the unit tests, describes the test as a string, and provides a function to perform the actual test. The **expect** method indicates whether the tests have succeeded.

Summary

Much of the test environment is set up for you when you use the Angular CLI. In the remainder of the book, we are going to focus on how use the lint checker, run the unit tests, and perform the end-to-end testing of the entire application. You can build an Angular application and never use these testing options; however, the benefits and confidence of automated testing processes should encourage you to make these tools part of your standard development life cycle.

Chapter 2 Lint Checking

What is wrong with the TypeScript code in Listing 3?

Code Listing 3: Simple TypeScript example

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

export class AppPage {
  navigateTo() {
    return browser.get(browser.baseUrl) as Promise<any>;
  }
  getTitleText(releaseNo : number) {
    let subTitle = '';
    switch (releaseNo) {
      case 1, 2:
        subTitle = 'Prior';
        break;
      default:
        subTitle = 'Current';
    }
    return subTitle;
  }
}
```

While there is nothing syntactically wrong with it, there are a few items that might cause problems. For example, you should declare the access level (public, private, or protected) on the method names. Using `<any>` somewhat defeats the purpose of a strongly typed language.

The more subtle issue is that if the `releaseNo` parameter contains the number 1, the code will still return `Current`. It is very likely this behavior isn't what the developer expects.

This is what the lint checker is all about—looking for code that compiles, but might have issues. You can decide whether the issue being reported is something you want to address, or if it's okay to leave the code alone.



Tip: The problem with the release number code is that the comma operator returns the last value in the list (in this example, the number 2). So, the value 1 will not trigger the first case, and will fall through to the default.

Installation

tslint will be installed as part of the Angular CLI installation, but you can also install it yourself using NPM. To install it globally, use the following command:

```
npm install tslint typescript -g
```

Although our focus in this book is Angular, you can use **tslint** for any TypeScript or JavaScript application.



***Note:** There is a similar linting tool named **ESLint** that some developers prefer. The **TSLint** tool is described in Chapter 3.*

TSLint options

The **tslint** command has several options you can use when running tslint. You can use the full option name (**-version**, or the abbreviated single letter where indicated).

-- **version** **-v**: This returns the version number, the current major version (at the time this book was written) is 5, with a minor version of 20.

-- **config** **-c**: This option allows you to indicate where the **tslint.json** configuration file to use can be found. If you do not specify a location, the first **tslint.json** file found in the path will be used. Within an Angular project, the configuration file is typically found in the project root folder.

-- **exclude** **-e**: This option allows you to exclude a file or folder from being lint checked. The **node_modules** folder contains thousands of third-party libraries and should be excluded from lint checking.

-- **fix**: Lint checking reads your files and produces warnings and suggestions. The -- **fix** option will automatically fix any found suggestion, if the lint rule supports automatic fixing. This means that your source code files most likely will be updated.



***Tip:** Be careful using this option—remember that lint checking offers suggestions, and it could fix something that could potentially break your code.*

-- **init** **-i**: Generates a default **tslint.json** config file in the current directory.

-- **project** **-p**: Specifies a path containing a **tsconfig.json** file that should be used to determine which files should be lint checked.

Other options

You can use the -- **help** options to see the complete set of **tslint** command-line options.

Return code

If you run lint checking as part of a continuous integration process, you might want to check the return code to decide if checking has passed or not. **tslint** returns the following values:

- 0: Linting produced no errors (although might have warnings).
- 1: Invalid arguments passed.
- 2: Linting failed with at least one rule with an error severity.

Configuration

Within an Angular project structure, you will find a file called **tslint.json**. This file contains the various rules you want the lint check to use when reviewing your source files.

Presets

Presets are predefined suggested sets of rules that you can use as a starting point for your lint checking. You can specify a preset using the **extends** keyword:

extends: “**tslint.recommended**”

The three defined presets are:

- **tslint.recommended:** This is the default generated by Angular/CLI, a stable set of rules for TypeScript programming.
- **tslint.latest:** This preset extends the recommended preset and is updated to include configuration for new rules added with each tslint release. For example, the comma operator example from Listing 3 was added during the tslint 5.8 release. It is not included in **tslint.recommended**, but appears in **tslint.latest**, as shown in the following code snippet:

```
// added in v5.8
"ban-comma-operator": true
```

Since **tslint.latest** is updated with new minor releases, you might see additional lint errors as new features are added. You might want to use **latest** for your local lint testing, but use the stable **recommended** for any automated build pipelines.

- **tslint.all:** This preset turns every rule to its strictest setting. Only for the very bravest among us.

You can visit [this website](#) to see the definition for each of the three presets.

Rules

The rules make up most of the configuration settings, and you can configure the settings to fine-tune how you want your code to be checked. For example, the default generated **tslint.json** file would not have caught any of the issues in Code Listing 3. In this section, we will highlight some of the various rules by category. You can see the complete list of code rules [here](#).

The general format is the rule name (generally quoted), followed by one or more parameters. For example, the rule to prevent the comma error from being reported is:

```
"ban-comma-operator": true
```

Some rules may have additional parameters. For example, if you wanted to restrict the number of lines allowed in a file to 150 lines, the syntax will be:

```
"max-file-line-count": [true,150]
```

TypeScript-specific

These rules are unique to TypeScript (since **tslint** has a JavaScript linter as well).

ban-ts-ignore

The TypeScript compiler will warn about unreachable code and other errors. By adding a comment with the **@ts-ignore:** syntax, you are telling the linter to not issue warnings. While a matter of preference, I generally like the linter to give me warnings, so I generally include this check.

no-any

Since TypeScript adds types to JavaScript, it defeats the purpose of typing if you declare your variable using the **Any** type. The **no-any** setting allows me to not accept non-typed variables.

no-magic-numbers

I personally dislike magic numbers in my code. For example, what does the following code mean?

```
if (result_code == 413)
```

Of course, you could look up the HTTP code for 413, if you know the **result_code** variable represents a status code. But if the code read:

```
if (result_code == REQUEST_TOO_LARGE)
```

You, and people maintaining the code, would have a much easier time reading the code. For this reason, I typically turn this error rule on.

Functionality

Certain code elements and statements might not be desirable in your code base, such as debugger statements and `eval`.

curly

Conditional and looping statements do not always require curly braces, but in general, using them can reduce the likelihood of subtle coding errors. By setting **curly** to true, you can identify expressions without curly braces as errors. You can provide additional parameters to tweak behavior a bit:

- **ignore-same-line**: Does not require braces when on the same line.
- **as-needed**: Only enforces rules if needed (such as, if only one expression, does not flag it as an error).

no-eval

The `eval` statement takes a string parameter and executes it as JavaScript code. This can be very dangerous, since there is no way to secure the string contents. In general, `eval()` should be avoided. The no-eval rule will flag as an error any use of the `eval()` function.

no-var-keyword

The `var` keyword is used to declare a variable, but the scope of the variable is to the function body, not the enclosing block. In addition, using `var` at the top level will create a property on the global object. In general, it is better programming practice to keep the variable scope as small as possible. The `let` and `const` statements keep the scope local to the enclosing block, and do not add the variable as a property on the global object. The no-var-keyword rule reports as an error any use of the `var` command to declare a variable.



Tip: *TSLint can fix `var` declarations by using either `let` (variable can be edited) or `const` (variable value is not changed)*

Maintainability

The maintainability rules are used to make your code easier to maintain (for you or future developers). Two rules I generally add are described in this section.

cyclomatic-complexity

This is a code metric as to how complex (confusing) a routine appears to be. The score starts at zero, then is incremented for decision statements that can add to the control flow. A common saying is that it's "harder to read code than to write it." Often, developers will rewrite a piece of code, rather than trying to understand what the code is actually doing.

System design should have small, single-purpose functions with meaningful variable names, which your future self or other developers will appreciate.

You can turn complexity measurement on, and set the maximum allowed score, using the following rule:

```
“cyclomatic-complexity”: [true,15]
```

This will compute the complexity and report an error if the score exceeds 15. Scores of 1–10 indicate good, easy-to-test code. Code with scores above 20–25 become more complex and are more difficult to test. A complexity score also suggests how many tests you need to write to completely test the function. You can read more about code complexity [here](#).

deprecation

The `@deprecated` tag in a comment is used to indicate a function or module has been deprecated. By setting this rule to true, such code will be flagged as linting errors.

max-classes-per-file

This rule controls how many classes can be defined in a file, and is set using the array structure of `[true, number of classes]`. I generally program one class per file, but you can use this option to set your own standards. Similarly, this is a max-lines-per-file rule, as mentioned in an example at the beginning of this chapter.

Style

Style rules are generally applied to encourage developers to write consistent code with other developers. You can control things like comment rules, header format, and file naming.

file-name-casing

This setting allows you to specify the casing rule (camel-case, pascal-case, kebab-case, snake-case) for all file names, or you can set specific casing rules based on file extension. The rule format is:

```
“file-name-casing”: [ true, “pascal-case”]
```

This causes all file names that are not in Pascal case to be flagged as a lint error.

You can also pass an array to use different casing by file extension:

```
“file-name-casing”: [ true, {“.css”: “pascal-case”,“.ts”: “camel-case”} ]
```

The ignore option causes certain file extension to ignore casing rules.

one-variable-per-declaration

A style preference some developers like is to have each variable declaration on its own line. You can set this rule to true to require each variable declaration (`let` or `const`) have its own line. This would prevent statements such as:

```
const salesTaxRate = 0.2, incomeTaxRate = .25
```

Instead, requiring:

```
const salesTaxRate = 0.2;
const incomeTaxRate = 0.25;
```

variable-name

There is an old saying that there are only two hard things in computer science: cache invalidation and naming things. While that may be true, and it is hard to assume everyone on the team will use meaningful variable names, we can at least use **tslint** to make the variable names look consistent.

The variable name rule is an array of options, controlling how variables can be named.

```
"variable-name": {
  "options": [
    "ban-keywords",
    "check-format",
    "allow-leading-underscore"
  ]
}
```

ban-keywords prevents using various TypeScript keywords (**any**, **number**, **string**, **boolean**, **undefined**) for variable or parameter names.

check-format uses lower camel case names for variables and uppercase for constants.

With **allowing-leading-underscore**, a variable name can begin with an underscore.

You can also use **allow-pascal-case** and **allow-snake-case** to increase the allowed case rules for variables (these are added to rule check).

Hopefully, as AI improves, we can add a new rule: use sensible variable names.

Format

The format rules generally deal with things like punctuation, white space, and line length. The goal of these rules is to make the code easier to read by keeping its appearance consistent. Most of the rules in this group can be automatically fixed by the lint process.

max-line-length

This setting allows you to control how long an individual source code line should be. It takes an array with two values: **true** and the line length.

```
"max-line-length": [true,100]
```

There are additional options to fine-tune which lines should be checked for a maximum line length, such as **ignore-pattern**. This allows you to specify regex patterns that, if found, cause the line length check to be ignored.

semicolon

The semicolon rule determines whether semicolons should be added at the end of statements. It is set using the following syntax:

```
"semicolon": [ true, "always" | "never"]
```

If you choose **always**, the semicolon must be present, even if not technically required. The **never** option does not allow semicolons unless they are required by the statement.

whitespace

White space can improve code readability. The rule is set by defining an array with **true** and any number of options.

```
"whitespace": [ true, "check-branch", "check-decl", "check-operator" ]
```

"check-branch" looks for white space around branching statements (**if**, **else**, **while**).

"check-decl" makes sure there is white space around the equals assignment, for example, **taxRate = 0.25**.

"check-operator" makes sure there is white space around operator tokens, for example, **rate = rate + 5**.

There are other settings available to tweak how white space is handled in your code.

Fixers

Some of the code can be fixed as part of the lint process. The rules will have the following icon (Figure 1) attached to them on the tsLint website.



Figure 1: Has fixer

You can run a lint check with the **-fix** option if you want some of the errors to be automatically corrected. However, you should back up your files to be safe, just in case a "fixed item" breaks your compile. I would suggest running the fixer option on a single file at a time, and only if there are too many items to fix manually.

Code Listing 4 shows our routing modules and the lint errors.

Code Listing 4: Lint check code

```
const routes: Routes = [
  { path: 'withdrawal', component: WithdrawalComponent },
  { path: 'deposit', component: DepositComponent },
  { path: "transfer", component: TransferComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
  public feePercent = 0.025
}
```

Two errors are reported: the double quotes on the **transfer** path and the missing semicolon in the **feePercent** assignment. (There is also third error, trailing white space, which is hard to illustrate in a book format.)

Code Listing 5 shows the code snippet after running the TSLint **-fix** option on this file.

Code Listing 5: Automatically fixed rules

```
const routes: Routes = [
  { path: 'withdrawal', component: WithdrawalComponent },
  { path: 'deposit', component: DepositComponent },
  { path: 'transfer', component: TransferComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
  public feePercent = .025;
```

Notice that the quotes have been changed on the **transfer** path, and the semicolon has been added after the **feePercent** assignment statement. The trailing white space has also been corrected.

Fixers are more common in the style and maintainability sections, but still should be run with care, since your source code will be modified, and the process will not back up your file.

Summary

Linting is a great, quick tool to find those little gotchas that the compiler is okay with but might not be what you wanted. It can also be very useful in a team environment to ensure the code is consistently styled and maintainable. I would recommend requiring that lint checks pass prior to all code check-ins, and including lint checking in your continuous integration environment.

Chapter 3 Other Linters

TSLint is a powerful linting option, and the default installed by Angular. However, there are other linters to consider. While TSLint will still be available, the author has indicated the program will be deprecated, and development on ESLint will be his priority. You can still use TSLint, but at some point, Angular may switch to ESLint. In this chapter, we will cover installing ESLint and using it in an Angular application, and we will look at SonarLint, a linting tool you can integrate into most IDEs.

SonarLint

SonarQube is a static code analysis tool that provides extremely thorough code analysis. Lint checking generally works on a statement level, for example, to report use of the debugger keyword, or to require lines end with a semicolon. There are some features that will check a component module, such as member ordering (how fields and methods are ordered in a component), but for the most part they focus on style rules, formatting, maintainability, and so on.

SonarQube is more focused on the overall, looking at things like function size (functions shouldn't be too big), credentials (should not be hard-coded), and commented-out code (comments should not contain code). Adding SonarLint (from SonarQube) to your lint process can identify issues beyond what the lint checker detects.

Installing SonarLint

You can search for SonarLint in the Visual Studio Marketplace, or find it directly at [this link](#).

Once you install it, you might be prompted to install a Java runtime. SonarLint should find the Java runtime automatically, or you can add the location in your Visual Studio Code settings.

```
{
  "sonarlint.ls.javaHome": "C:\\\\Program Files\\Java\\jre1.8.0_131"
}
```

You can visit [this link](#) if you want to install it in other code editors or integrated development environments. The top of the page lists other IDEs SonarLint can be integrated with.

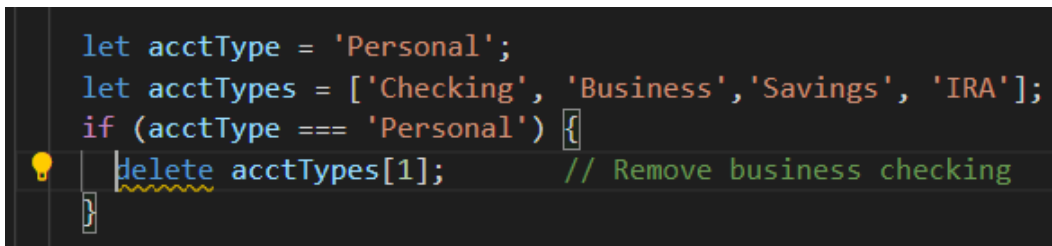
SonarLint rules

SonarLint has over 100 TypeScript rules, and they are classified as code smells, bugs, and security hotspots.

Code smells

Code smells are items that technically will compile, but possibly point to a problem. For example, duplicated code in multiple locations will compile but cause issues if the code needs to be updated and one or more of the locations is not updated. This bug would be very hard to catch.

Figure 2 shows an example code smell identified by SonarLint.

A screenshot of a code editor showing a TypeScript snippet. The code defines an array 'acctTypes' with values 'Checking', 'Business', 'Savings', and 'IRA'. An if-statement checks if 'acctType' is 'Personal'. Inside the if-block, the command 'delete acctTypes[1];' is used to remove the 'Business' element. A light bulb icon is shown next to the 'delete' command, indicating a code smell. A tooltip is visible over the light bulb, showing the text 'Use "indexOf" or "includes" (available from ES2016) instead. sonarlint(typescript:S4619)'.

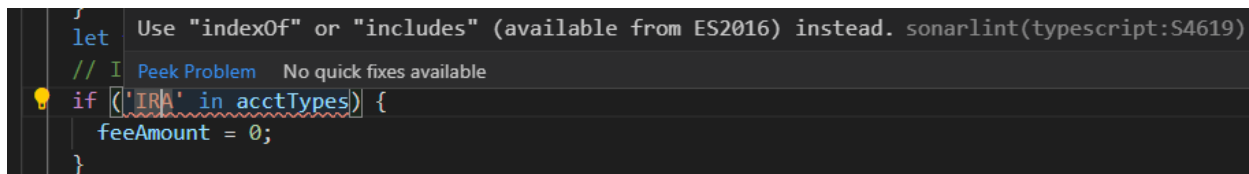
```
let acctType = 'Personal';
let acctTypes = ['Checking', 'Business', 'Savings', 'IRA'];
if (acctType === 'Personal') {
  delete acctTypes[1]; // Remove business checking
}
```

Figure 2: SonarLint code smell

By right-clicking on the light bulb icon, you will be able to bring up the related help or disable the rule itself. In the example shown in Figure 2, the `delete` command replaced the array element with undefined, rather than removing the element from the array (which `array.splice` will do).

Bugs

Bugs are code elements that compile but won't do what the code suggests. Figure 3 shows an example of a bug detected by SonarLint.

A screenshot of a code editor showing a TypeScript snippet. The code defines a variable 'feeAmount' and sets it to 0 if 'IRA' is in the 'acctTypes' array. A light bulb icon is shown next to the 'in' operator in the if-statement. A tooltip is visible over the light bulb, showing the text 'Use "indexOf" or "includes" (available from ES2016) instead. sonarlint(typescript:S4619)'.

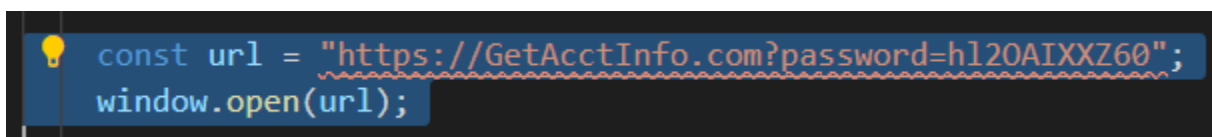
```
let feeAmount = 0;
if ('IRA' in acctTypes) {
  feeAmount = 0;
}
```

Figure 3: SonarLint bug

The `in` command works on array indexes, not values. SonarLint shows hint text with the suggested fix, and you can also right-click the light bulb for more details about the issue.

Security hotspots

Security hotspots identify areas of the code that pose a security risk. Figure 4 shows an example where the credentials were hard-coded, which SonarLint identified as a potential security concern.

A screenshot of a code editor showing a JavaScript snippet. The code defines a constant 'url' with a hard-coded password 'h120AIXXZ60' and uses 'window.open(url)'. A light bulb icon is shown next to the 'url' variable, indicating a security hotspot.

```
const url = "https://GetAcctInfo.com?password=h120AIXXZ60";
window.open(url);
```

Figure 4: SonarLint security hotspot

Real-time checking

Some of the issues raised by SonarLint may also be caught by the lint checker, but installing the SonarLint extension into your IDE will help you catch the errors while coding, so they can be addressed prior to the lint step.

ESLint

ESLint is the linter package that the TypeScript development team has decided to use. In some future Angular release, it is possible that the CLI will generate the lint configuration to use ESLint instead. You can add ESLint to your current configuration (while keeping TSLint) if you want to get familiar with this linter.

Install ESLint

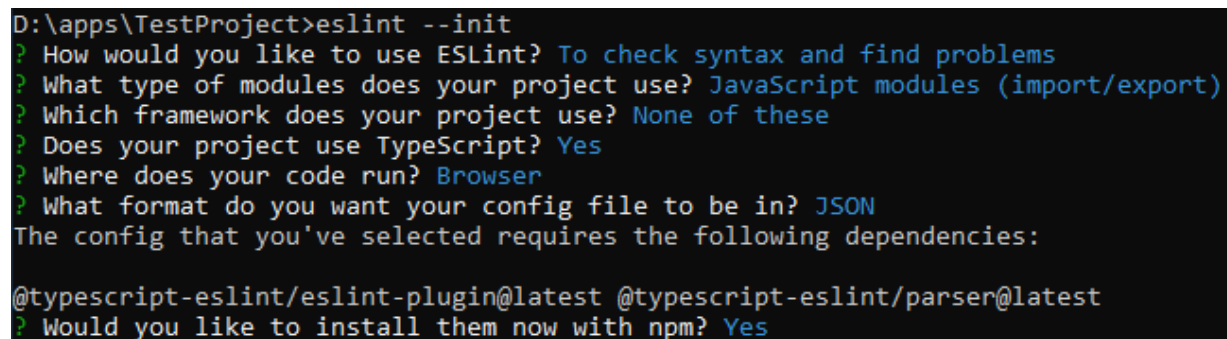
The first step is to install the ESLint package using npm. The syntax is:

```
npm install eslint --save-dev
```

Once you've installed ESLint, you will need to create a configuration file using the command syntax:

```
eslint --init
```

This command will lead you through a series of questions and then create the **.eslintrc.json** file, which contains the ESLint configuration options. Figure 5 shows the questions.



```
D:\apps\TestProject>eslint --init
? How would you like to use ESLint? To check syntax and find problems
? What type of modules does your project use? JavaScript modules (import/export)
? Which framework does your project use? None of these
? Does your project use TypeScript? Yes
? Where does your code run? Browser
? What format do you want your config file to be in? JSON
The config that you've selected requires the following dependencies:
@typescript-eslint/eslint-plugin@latest @typescript-eslint/parser@latest
? Would you like to install them now with npm? Yes
```

Figure 5: Creating ESLint config file

Once you've installed ESLint and created the configuration file, you can run the following command to use it against your files.

```
eslint ./src/**/*.ts --ext .ts --ignore-pattern **/*.spec.ts --fix-dry-run
```

The command looks for all **.ts** files in the indicated directory, ignoring the spec files (testing source files). We also asked it to show us any fixes it could potentially apply. If you run **eslint** by itself, you can get a complete list of command-line parameters.

You can also add the output information if you want to save the lint output to a file. For example, adding the following code will save the output to a JSON file called **lint.json** in the current directory:

```
--format json --o lint.json
```

This can be handy if you want some sort of automated CI process to check for linting issues.

eslint output

Figure 6 shows a sample output from **eslint**.

```
D:\apps\TestProject>eslint ./src/**/*.ts --ext .ts --ignore-pattern **/*.spec.ts --fix-dry-run
D:\apps\TestProject\src\app\app-routing.module.ts
  2:10  error  'Routes' is defined but never used          no-unused-vars
  6:10  error  'stringify' is defined but never used       no-unused-vars
 26:5   error  The update clause in this loop moves the   for-direction
       error  variable in the wrong direction            no-empty
 33:22  error  Empty block statement
D:\apps\TestProject\src\app\deposit\deposit.component.ts
  1:21  error  'OnInit' is defined but never used         no-unused-vars
D:\apps\TestProject\src\app\transfer\transfer.component.ts
  1:21  error  'OnInit' is defined but never used         no-unused-vars
D:\apps\TestProject\src\app\withdrawal\withdrawal.component.ts
  1:21  error  'OnInit' is defined but never used         no-unused-vars
7 problems (7 errors, 0 warnings)
```

Figure 6: *eslint* output example

It is interesting to note that **tslint** gave the app-routing module no errors, but **eslint** detected the unused **stringify** command and the loop variable error in a **for** loop. This is not to say one is better than the other, but that the tools can be used together to analyze and improve your software.

Summary

In this chapter, we looked at SonarLint and ESLint, two additional tools to consider adding to your Angular application development environment. See [Chapter 9](#) to find out more about these two tools.

Chapter 4 Test-Driven Development

Test-driven development is a software development technique in which the requirements are written as test cases, and the software is written/refactored to pass the various test cases.

ATM testing

An automated teller machine (ATM) is being created, and we are designing the software for it. We plan on a method called **ShouldCashBeDisbursed()** and expect a Boolean value to be returned. Code Listing 6 shows our initial code.

Code Listing 6: ShouldCashBeDisbursed function

```
public ShouldCashBeDisbursed(balance: number , withdrawal: number ) {  
    let ans = null;  
    return ans;  
}
```

At this point, our code does nothing except return a NULL. If you had created an interface, this code might be the default implementation for one of the definitions.

We start by writing our test cases, which we'd expect to all fail. Listing 7 shows our test cases.

Code Listing 7: Test cases

```
it('should dispense $200 when balance = $1000', () =>  
{  
    let ans = component.ShouldCashBeDisbursed(1000,200);  
    expect(ans).toEqual(true);  
});  
  
it('should not dispense $200 when balance = $150', () =>  
{  
    let ans = component.ShouldCashBeDisbursed(150,200);  
    expect(ans).toEqual(false);  
});
```

We now write the code necessary to pass these tests. However, as you write the tests, you will likely think of additional tests you might want to add. What happens if the requested withdrawal amount is 0? Or a negative number? Thinking through your test steps can help you write a more robust method.

Once you've written your tests, you can go ahead and code the method. In addition to helping you think of your design, you've also built a regression test plan to ensure future changes to the method do not break previous features. Code Listing 8 shows our final test steps.

Code Listing 8: Withdrawal test cases

```
it('should dispense $200 when balance = $1000', () =>
{
  let ans = component.ShouldCashBeDisbursed(1000,200);
  expect(ans).toEqual(true);
});

it('should not dispense $200 when balance = $150', () =>
{
  let ans = component.ShouldCashBeDisbursed(150,200);
  expect(ans).toEqual(false);
});

it('should not dispense when withdrawal is negative', () =>
{
  let ans = component.ShouldCashBeDisbursed(1000,-200);
  expect(ans).toEqual(false);
});

it('should not dispense when withdrawal is 0', () =>
{
  let ans = component.ShouldCashBeDisbursed(150,0);
  expect(ans).toEqual(false);
});
```

Code Listing 9 shows our method.

Code Listing 9: AllowWithdrawal method

```
public ShouldCashBeDisbursed(balance: number , withdrawal: number ) {
  const shouldAllow =
    (balance - withdrawal > 0) && (withdrawal>0)
  if (shouldAllow) {
    balance = balance - withdrawal;
  }
  return (shouldAllow);
}
```

When we run this in the test runner (described in later chapters), the result is our expected passed tests. Figure 7 shows the test runner results.

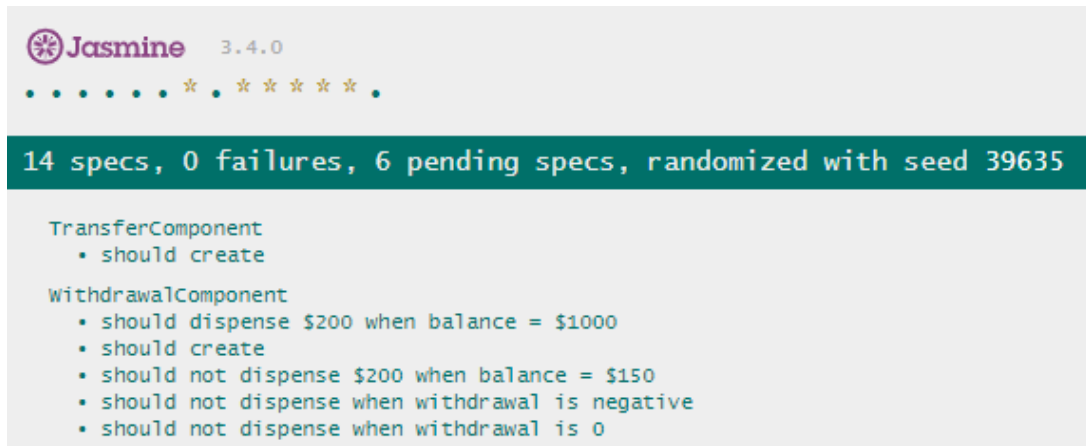


Figure 7: Test results

Behavior-driven development

While test-driven development works, it is generally focused at the method level. This is fine for developers, but not necessarily for business analysts. They are more concerned with the behavior of the system, not the details of the methods. This is the focus of behavior-driven development (BDD). The business analysts write the expected behavior, which acts as a specification and a test plan for the development team.

Let's look at how business analysts might expect the ATM withdrawal function to work. The business analyst will think in terms of the user and create user stories. The user story for the ATM might look like:

- **As a:** customer using the ATM
- **I want:** to be able to withdraw cash from my account
- **So that:** I can take my spouse to dinner.

Using the [Gherkin language](#), the business analyst will then write examples of how they expect the system to behave. For example,

- **Given** I am using the ATM machine
- **When** I withdraw \$200 with a balance of \$1000
- **Then** I expect to be dispensed \$200

And:

- **Given** I am using the ATM machine
- **When** I withdraw \$200 with a balance of \$150
- **Then** I expect to be shown an "insufficient funds" message

So far, this is good, but now they add a couple more scenarios:

- **Given** I am using the ATM machine
- **When** I withdraw \$0
- **Then** I expect to be shown a "Please enter a valid amount" message

And:

- **Given** I am using the ATM machine
- **When** I withdraw more than the \$300 daily limit
- **Then** I expect to be shown a “Cannot withdraw more than \$300” message

Now that the business analysts have written their expected behavior out, the developer realizes that a Boolean return value won't work. The method needs to return a status code, since the error message needs to be different. Also notice that the business analyst never considered the negative amount case, since most ATM machines don't have + or – buttons.

Summary

When the tests are written first, it forces more thinking and designing, instead of jumping in and coding right away. Whether the tests are written by developers or business analysts, the design work of writing the test plan first has several benefits:

- Encourages thinking before coding!
- Gives you a regression test base, ensuring nothing breaks after code changes.
- Forms a contract between the specifications and the code.

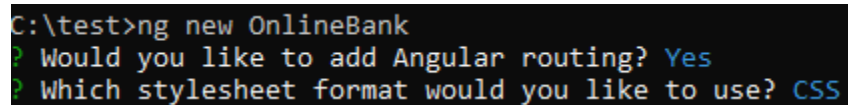
There are many available tools for and variations of test-driven development. In the next few chapters, we will explore the Angular environment for working with this approach.

Chapter 5 Angular CLI

When you use the Angular CLI application to generate a project and components, it will generate your code and set up testing files for you. We are going to create a sample application of an online bank for later chapters. Let's look at the basic setup using the CLI.

New project

Start a new project using the Angular CLI, as shown in Figure 8.



```
C:\test>ng new OnlineBank
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
```

Figure 8: Angular CLI new project

If you open the project folder (**OnlineBank**), you will see several configuration files, the node modules folder, and your **src** code folder. There will also be an **e2e** folder for end-to-end testing.

Within the **src\app** folder is the actual application code. The app.component has four files generated. The .html file is the template code, and the .css file is your style sheet (the extension might vary depending on your selected style sheet format). The .ts file (shown in Code Listing 10) is the actual component code that ties the pieces together.

Code Listing 10: Default component

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'OnlineBank';
}
```

The **.spec.ts** file is the unit test file generated by the Angular CLI. It contains the necessary code to start your unit testing. It simply confirms the component can be created, and the title is OnlineBank, as expected.

Create a service

Our next step is to navigate to the project folder and create a service to get information about the customer (name, PIN number, balance, etc.). Figure 9 shows the command line to generate the service code.

```
C:\test\OnlineBank>ng generate service
? What name would you like to use for the service? CustomerInfo
CREATE src/app/customer-info.service.spec.ts (364 bytes)
CREATE src/app/customer-info.service.ts (141 bytes)
```

Figure 9: Create a service

Note that in addition to the service, the CLI also generated the unit test in the **.spec.ts** file. Since the CLI doesn't know anything about the service, it will simply generate the structure and an empty constructor. The spec file simply tests that the service can be created. Code Listing 11 shows the sample spec file generated for the service.

Code Listing 11: Service test spec file

```
import { TestBed } from '@angular/core/testing';
import { CustomerInfoService } from './customer-info.service';

describe('CustomerInfoService', () => {
  beforeEach(() => TestBed.configureTestingModule({}));
  it('should be created', () => {
    const service: CustomerInfoService = TestBed.get(CustomerInfoService);
    expect(service).toBeTruthy();
  });
});
```

We will explore these spec files in [Chapter 7, “Unit Testing.”](#)

Create a component

Figure 10 shows our command to generate a withdrawal component. The component source file and testing files are generated for us, and the **app.module** file is updated to import and declare the new component.

```
C:\test\OnlineBank>ng generate component
? What name would you like to use for the component? Withdrawal
CREATE src/app/withdrawal/withdrawal.component.html (25 bytes)
CREATE src/app/withdrawal/withdrawal.component.spec.ts (656 bytes)
CREATE src/app/withdrawal/withdrawal.component.ts (285 bytes)
CREATE src/app/withdrawal/withdrawal.component.css (0 bytes)
UPDATE src/app/app.module.ts (491 bytes)
```

Figure 10: Generate withdrawal component

At this point, we have the basic starting point and all testing files for our application. There is also an **e2e** directory in the project root folder. This folder will contain a starting application for our end-to-end test process (which we cover in [Chapter 8, “E2E Testing with Jasmine”](#)).

Summary

Using the Angular CLI gives you all the necessary files for your application and testing work. As we explore the testing tools, we will take these generated files and expand them to provide a testable application.

Chapter 6 Sample Application

For the chapters on unit testing and end-to-end testing, we are going to use a sample Angular application called Online Bank. This application acts as an online ATM. It is meant to provide a simple service and a couple components so we can see how testing is used in an actual example program.

Main screen

This is a sample screen from our test application. It will not win any UX awards, but we can use it and the source code to create unit and E2E tests. The initial screen simply prompts the user to enter a PIN. The menu options are not visible until the user enters their correct PIN.

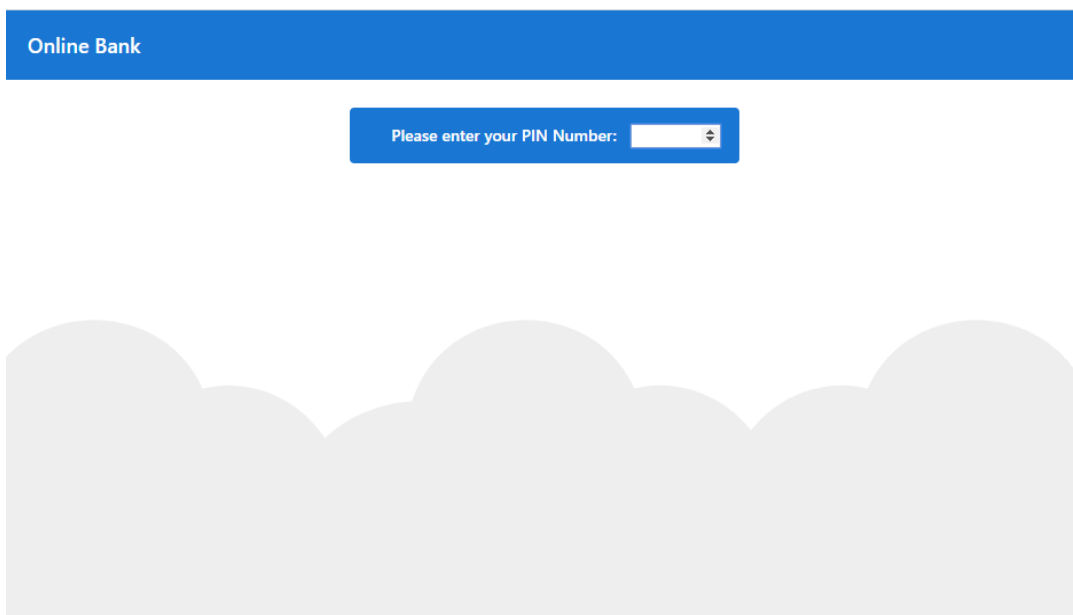


Figure 11: Main ATM screen

Once the user enters their PIN, the application will show the menu options and the user's name and balance, as shown in Figure 12.

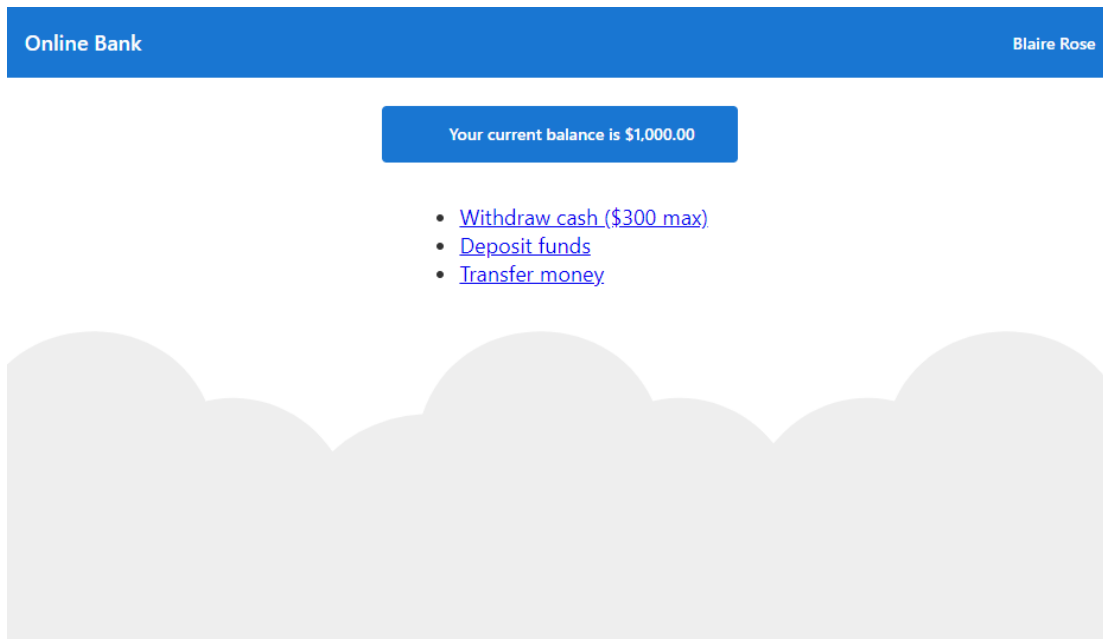


Figure 12: Main menu

Application files

The application consists of a component, a service, and routing module, as well as three component modules:

- app.module
- app-routing.module
- app.component
- customer-info.service
- deposit\deposit.component
- transfer\transfer.component
- withdrawal\withdrawal.component

App.module and **app-routing.module** will not need to be tested; they are setup work for the Angular application. **App.component** is our main screen component, and has no actual component code, just some HTML template code. There is also the menu code, which is a series of router links.

Service

The **customer-info** service code module is a simple service that contains methods to get customer information and to update the balance and the PIN. The primary method **GetCustomerDetails()** gets parameters of the card number and a PIN value. In the actual application, the card number would be read by the machine when the user inserts the card into the ATM. They would then key in the PIN for two-factor authentication (something we have and know).

Code Listing 12 shows the simple customer info service. Typically, this would require an Ajax call to some web service to provide the requested information.

Code Listing 12: Customer-info service

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
export class CustomerInfoService {
  private Customers = [
    { id: 1, cardNo: '1234', name: 'Blaire', balance: 1200, pin: '0825' },
    { id: 2, cardNo: '5678', name: 'John', balance: 600, pin: '1224' },
    { id: 3, cardNo: '8888', name: 'Kaia', balance: 2500, pin: '0322' },
    { id: 4, cardNo: '9999', name: 'Jonathan', balance: 500, pin: '0604' },
    { id: 5, cardNo: '0000', name: 'Kellie', balance: 4500, pin: '0304' }
  ];
  constructor() { }
  public GetCustomerDetails(cardNumber: string, pinNumber: string) {
    // Typically, an Ajax call to get information about customer
    const x = this.Customers.findIndex((x) => x.cardNo ===
                                         cardNumber && x.pin === pinNumber)
;
    let details;
    if (x >= 0) {
      details = this.Customers[x];
    }
    return details;
  }
  public UpdateBalance(id: number, amt: number) {
    let newBalance = NaN;
    if (id >= 0 && id < this.Customers.length) {
      this.Customers[id].balance += amt;
      newBalance = this.Customers[id].balance;
    }
    return newBalance;
  }
}
```

The method returns either an undefined value, or a simple object with customer information if a valid card number and PIN are provided. A unit test for this service would require at least two tests: one to confirm a valid object with a good card and PIN, and a second to test for an undefined object if invalid credentials are supplied.

Components

Our three component modules (deposit, transfer, and withdrawal) are the other primary targets of our unit testing code. For example, the following two methods in the **withdrawal.component** are the focus of our unit test for that component. Code Listing 13 shows these two methods.

Code Listing 13: Withdrawal methods

```
public ShouldCashBeDisbursed(withdrawal: number) {
    const shouldAllow = this.balance - withdrawal > 0;
    if (shouldAllow) {
        this.balance = this.balance - withdrawal;
        this.service.UpdateBalance(withdrawal * -1);
    }
    return (shouldAllow);
}
public AssessLowBalanceFee(minBalance: number) {
    if (this.balance < minBalance) {
        this.balance = this.balance - 25;
        this.service.UpdateBalance(-25);
    }
}
};
```

You will see that the test specification files (in the next chapter) contain test cases. Before reading further, think of the scenarios you should test for those method calls.

What should we test?

One of the key things to consider when designing unit tests is what we want to learn from the tests. If we look at our main app component, it is a basic, empty component with an associated HTML file. The component has no methods, so we could write some unit tests for the generated webpage. For example, here are a few tests we could write:

- Should have as title “Online Bank.”
- Should only have three menu actions.
- Withdrawal link should say “Withdraw cash (\$300 max).”

These tests will simply confirm that the generated webpage contains the expected text. If multiple developers can edit the source file, and there is no code review, these tests could be useful to detect if a change was made to the source code. However, you need to determine if your development environment needs such testing. In general, you shouldn't need unit tests for static HTML pages. Version control systems and reviewed check-in policies are better solutions for detecting source code changes.



Note: *In our example menu, we've hard-coded the maximum limit as a string. If the rules change and more cash can be withdrawn, this could trigger an error during an E2E test (if the business analysts updated the specs to reflect the new limit).*

Component-generated HTML

If your component generates HTML, possibly from an Ajax call or other manipulation, then testing the generated webpage makes sense in a unit test. It confirms that the actual code to generate the page performs as expected. In our code, once the user enters a valid PIN, we'd expect their name to appear in the banner. A test to confirm that the banner has a non-empty string after successful PIN entry is the type of behavior we should look for in our unit tests.

Component methods

The component methods should be the primary target of your unit tests. Look at the parameters to the method and try to create unit tests with both expected good values and problem values (for example, what would happen if I were to pass a negative number to my withdrawal method). Don't design tests with only good input; instead, try deliberately bad input. The mere thought of designing a test with bad input can often make your code better.

Looking at the `ShouldCashBeDisbursed()` method, what should be returned if I pass a negative number? Assuming the balance is positive, the method will always return true and will increase the balance. With this knowledge obtained by thinking about the unit test, we might decide that a Boolean return value is not the best, perhaps a status code or triggering an exception would be better. By thinking about your tests, you will find yourself writing more robust code.



Note: *TDD (test-driven development) is exactly about this—write your tests first, and they will all fail (since there's no code yet). As you implement your method and the tests begin to pass, you can be more confident in your method implementation than if you code first, test later.*

Summary

Our application is a simple example of an Angular application, so we can focus on how to create unit and end-to-end tests. You can find the source code [here](#) (with complete testing files).

Chapter 7 Unit Testing

Each component within an Angular application should have unit tests written to ensure the component performs as expected. When you're using the Angular CLI, unit test files will be written for you. These files (spec files) will provide the basic shell to initialize the component and run some simple tests the CLI can generate from the component. As you add functionality to the component, you should add more testing to the spec files to keep your unit test current with your component.

Unit tests should be small, testing a single function or method within your component. They should be independent, with a goal that if a test fails, you know exactly what piece of code to review or fix to make the test pass. If our component sorts a retrieved collection of names, we want to test the sorting code separate from the code that retrieves the collection.

The only testable goal is to confirm that the sort method sorts the collection; a separate test should ensure the collection is retrieved in sorted order. If we combine the retrieval and sort into a single test method, and the method fails, we do not know if the retrieval or the sort failed.

You want to design separate tests, one confirming the data is retrieved properly, and a second test confirming that sorting works.

Spec files

When components are generated for you, the CLI will generate a file with the same name as the component ending with `.spec.ts`. This file will import the necessary testing modules and write a basic set of unit tests. Typically, each component will have a single spec file associated with it, and this spec file will have multiple individual unit tests. Listing 14 shows a simple spec file and a unit test.

Code Listing 14: Simple spec file

```
import { TestBed, async } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  });

  it(`should have as title 'OnlineBank'`, () => {
    const fixture = TestBed.createComponent(AppComponent);
```



```
const app = fixture.debugElement.componentInstance;
expect(app.title).toEqual('OnlineBank');
});
```

TestBed

The **TestBed** object from the Angular testing module is used to allow us to create component instances. If we look at our test spec file for our simple service, you will see the **TestBed** object is being used to create an instance of the service. Code Listing 15 shows the service test file.

Code Listing 15: Service test file

```
import { TestBed } from '@angular/core/testing';
import { CustomerInfoService } from './customer-info.service';

describe('CustomerInfoService', () => {
  beforeEach(() => TestBed.configureTestingModule({}));
  it('should be created', () => {
    const service: CustomerInfoService = TestBed.get(CustomerInfoService);
    expect(service).toBeTruthy();
  });
  it('return customer information for valid PIN', () => {
    const service: CustomerInfoService = TestBed.get(CustomerInfoService);
    const ans = service.GetCustomerDetails('1111', '0825');
    expect(ans).toBeTruthy();
  });
  it('return undefined for invalid PIN', () => {
    const service: CustomerInfoService = TestBed.get(CustomerInfoService);
    const ans = service.GetCustomerDetails('1111', '1234');
    expect(ans).toBeUndefined();
  });
});
```

For services, the **TestBed** object simply needs to call the **get()** method to create an instance of the service. However, creating components using **TestBed** requires a bit more effort.

configureTestingModule()

In an Angular application, you will have an **NgModule()**, typically in the **app.module.ts** file. Code Listing 16 shows an example **NgModule()** setup.

Code Listing 16: NgModule

```
@NgModule({
  declarations: [
    AppComponent,
    WithdrawalComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [CustomerInfoService],
  bootstrap: [AppComponent]
})
```

The **configureTestingModule()** method in the **TestBed** object allows you to set up your module options in the testing environment. You can specify providers, declarations, and imports in an object collection passed to the method. The code snippet in Code Listing 17 shows the **TestBed** being configured before each test within our test file.

Code Listing 17: Configuring TestBed

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ WithdrawalComponent ],
    providers: [ CustomerInfoService ]
  })
  .compileComponents();
}));
```

This setup allows you to mimic **NgModule** as needed by your test application.

createComponent() method

To create the component, you need to call the **TestBed**'s **createComponent** method and specify the component you wish to create. The resulting object is a wrapper containing both the component and the template code. The actual component can be obtained via the component instance property of the **createComponent** object. Code Listing 18 shows example code to get the component from the **TestBed** object.

Code Listing 18: Getting a component from TestBed

```
fixture = TestBed.createComponent(WithdrawalComponent);
component = fixture.componentInstance;
```

You can test interactions with the component, calling its methods and looking at its properties.

Getting a service from TestBed

You can access the `TestBed`'s `get()` method to retrieve any provided services. For example, the following snippet would return an instance of the customer-info service in our application.

```
customerInfo = TestBed.get(CustomerInfoService)
```

`TestBed` provides the methods to mimic `NgModule` and create service and component instances with your Angular testing files.

Jasmine

Jasmine is a development framework for behavior-driven development. The goal of behavior-driven development is to allow business analysts, QA analysts, and developers to create a common understanding of how an application should work. This is done by providing a simple sentence describing a function and its expected outcome.

As an example, let's imagine software for an automated teller machine (ATM). To process a withdrawal, there are several conditions that need to be considered:

- Customer requests less funds than the current bank balance: Provide the funds.
- Customer requests more funds than the balance: Display "Insufficient funds" error.

The business analyst might write the following in a common language.

Withdrawal behavior

- If the balance is enough, the withdrawal should be permitted.
- If not enough funds, then the withdrawal should not be permitted.

From a development point of view, there is quite a bit of behind-the-scenes work that must be done. We must determine the customer information and balance, compare it to the requested amount, decide whether to disburse the funds, and finally, update the customer balance. However, from a unit test point of view, for this test, all we are concerned about is the software decision to disburse or not.

The Jasmine framework allows us to write a specification for the above rules using a few simple commands.

describe() method

The **describe** command is the starting point for the collected set of unit tests. It takes two parameters: a text description of the component being tested, and a function containing the tests to be run.

```
describe('Withdrawal behavior for ATM', () => {} )
```

it() method

The **it** command describes the actual test being performed. It takes one required parameter, a text description of the test, and an optional second parameter, the function to execute the test. There is a third, optional parameter, a timeout value for async operations. For our example, we might use:

```
it('should permit withdrawal if sufficient funds')
```

```
it('should not permit withdrawal if insufficient funds')
```

We could run our unit tests at this point (using **ng test**), and we would see the following result:

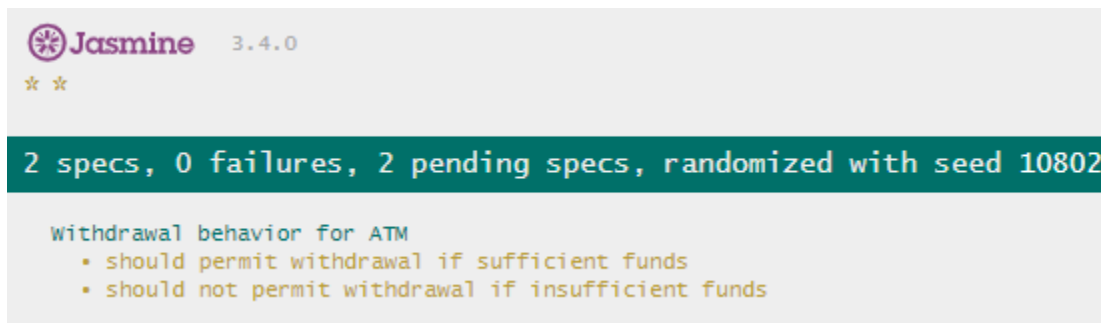


Figure 13: Unit test results

Because there is not a second parameter (function), Jasmine considers the test as pending. So, let's look at writing the function to perform the test. We start by writing the function to fail, as shown in Code Listing 19.

Code Listing 19: Failed unit test

```
it('should permit withdrawal if sufficient funds',() =>
{
  const decision = null;
  expect(decision).toBe(true);
});
```

If we run **ng test** now, this test will fail, since the expected value for decision is true. Actual test code will consist of the steps necessary to perform the named test and end with an expectation (**expect()** command).

To perform the actual testing, we will first need to get an instance of the component being tested. This is done in the following two lines. The first line creates the component object, and the second line provides us a debug instance of the component to work with.

```
const fixture = TestBed.createComponent(AppComponent);
const app = fixture.debugElement.componentInstance;
```

Once the component instance is created, we can interact with it and perform the test, as shown in Code Listing 20.

Code Listing 20: Unit test

```
it('should permit withdrawal if sufficient funds', () =>
{
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  app.balance = 2000; // Set the component balance
  const withdrawal = 200;
  let decision = app.ShouldCashBeDisbursed(withdrawal);
  expect(decision).toBe(true);
});
```

The component instance is created, the balance set, and the method called to determine if the withdrawal should be performed. The expectation is that with a balance of \$2,000, a withdrawal request for \$200 should be accepted. The unit test is simple and isolated. If the test fails, you should know exactly which method in the component code to review.

Code Listing 21 is the unit test for an unsuccessful withdrawal.

Code Listing 21: Insufficient funds

```
it('should not permit withdrawal if insufficient funds', () =>
{
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  app.balance = 500; // Set the component balance
  const withdrawal = 1200;
  let decision = app.ShouldCashBeDisbursed(withdrawal);
  expect(decision).toBe(false);
});
```

When you run the **ng test** command, the results will show both specifications passed.

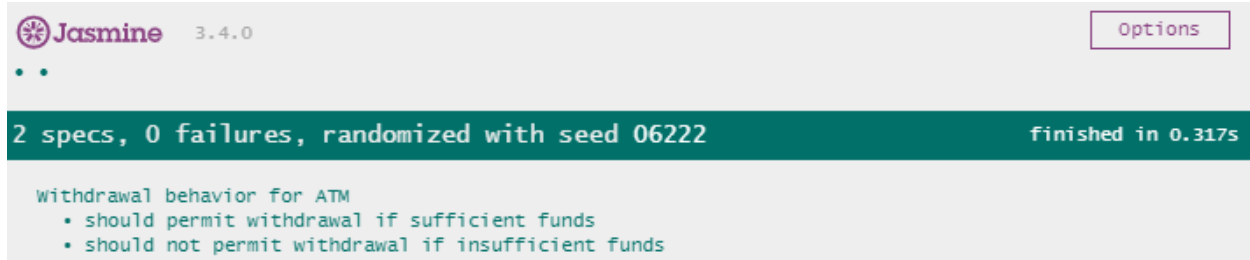


Figure 14: Passed unit tests

You should also always provide a test to confirm the component can be created, as shown in Code Listing 22.

Code Listing 22: Create component test

```
it('should create the app', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  expect(app).toBeTruthy();
});
```

If an error occurs during component initialization, then the test will fail, with the purpose of letting you know something is amiss during the constructor or `ng init` calls.

xit() method

The `xit()` method is the same syntax as the `it()` method; however, the `x` causes the method to be disabled. Figure 15 shows an example result when one of the methods is disabled:

```
Withdrawal behavior for ATM
  • should create the app
  • should have as title 'Withdrawal Component' PENDING WITH MESSAGE: Temporarily disabled with xit
  • should not permit withdrawal if insufficient funds
  • should permit withdrawal if sufficient funds
```

Figure 15: Disabled test method

fit() method

The `fit` (focus it) method tells the test running to only run the indicated tests. All other tests are skipped. These methods can be handy during the development phase to only focus on failing tests, without having to run the entire test suite. Figure 16 shows the result when a single method has focus.

```
Ran 1 of 4 specs - run all
Incomplete: fit() or fdescribe() was found, 1 spec, 0 failures, randomized with seed 60889

Withdrawal behavior for ATM
  • should have as title 'Withdrawal Component'
  • should create the app
  • should permit withdrawal if sufficient funds
  • should not permit withdrawal if insufficient funds
```

Figure 16: Single method has focus



Note: There are also *fDescribe* and *xDescribe* variations, in case you want to disable or focus an entire set of tests.

pending() method

The **pending()** method allows you to cause a test to be flagged as pending, and optionally provide a reason. The test will appear on the report as pending and provide the reason. Figure 17 shows how the code snippet would appear.

```
pending("Need to implement $5 fine to annoy customers");
```

```
Withdrawal behavior for ATM
  • should permit withdrawal if sufficient funds
  • should have as title 'Withdrawal Component'
  • should create the app
  • should not permit withdrawal if insufficient funds PENDING WITH MESSAGE: Need to implement $5 fine to annoy customers
```

Figure 17: Pending tests

expect() method

The **expect** method is your key to determining if the test passed or failed. For a test to succeed, all expected results must be true. While generally you'd have one expectation per test, you can have multiple **expect()** calls. For example, in our enough funds example, we might add an additional expectation.

```
expect(app.balance).toBe(1800);
```

However, unit tests should be small and isolated. If you find your unit test has lots of **expect()** calls, consider breaking it into additional tests.

The **expect** method takes a single parameter (the value being tested) and is followed by a chained list of matchers. Matchers are comparison operations to check if the parameter meets the expected criteria. Some common matchers are:

toBe(value)

Expects the parameter to be the exact value (uses **===** for comparison).

toBeCloseTo(value,precision)

When working with non-integer values (recall that TypeScript and JavaScript do not have an `int` data type), you might need to settle for close to value, rather than an exact match, so 49.999999 would be acceptable if your test expects a 50% value.

toBeTrue()

The parameter is expected to be a true value.

toBeFalse()

The parameter is expected to be a false value.

toBeTruthy() / toBeFalsy()

JavaScript and TypeScript consider the following values to be “falsy:”

- `false`
- `0`
- empty string (`''` or `""`)
- `null`
- `undefined`
- `NaN`

If the parameter is one of the above values, `toBeFalsy()` will match. If the parameter is any other value, then `toBeTruthy()` will match.

toBeLessThan(value), toBeGreaterThan(value)

These matchers test if a value is less than or greater than another value.

toBeLessThanOrEqual(value), toBeGreaterThanOrEqual(value)

These matchers test if a value is less than or equal to or greater than or equal to another value.

toContain(value)

If the parameter to `expect()` is an array, this matcher returns true if the value is found in the array. If the parameter is a string, this returns true if the substring is found within the parameter string value.

toMatch(regularExpression)

This powerful matcher allows you to compare a value using a regular expression. For example, you might use the following to check for a United States zip code.

```
Expect(zipCode).toMatch(/^\\d{5}(?:[-\\s]\\d{4})?$')
```


You can download my ebook [Regular Expressions Succinctly](#) if you want to learn more about regular expressions.

not

Negates another matcher: `not.toBe(1000)`.

There are quite a few additional matchers; you can find a complete list in [Appendix B](#).

Global commands

Jasmine has some additional methods to make unit test code a bit easier by allowing you to write code to be executed before and after the entire test suite and before and after each test.

beforeAll() method

This command takes a required parameter, a function to run prior to running the first test in the set of tests within the `describe()` function. A second parameter is a timeout value when using an async function.

For example, you might want to log to the console the version and platform the test is being performed on, as the following code snippet shows.

```
beforeAll( () => {  
  console.log("Starting Withdrawal Test");  
  console.log(navigator.appVersion+' on '+navigator.platform);  
});
```

Be careful however not to create shared information that could cause false results in your tests.

beforeEach() method

This method provides a function to be called before each individual test (`it()` method) in the described unit test. Most of the CLI-generated spec files will contain this method to build the testbed used by the unit tests. Listing 23 shows the typical `beforeEach` method.

Code Listing 23: beforeEach

```
beforeEach(async(() => {  
  TestBed.configureTestingModule({  
    imports: [  
      RouterTestingModule  
    ],  
    declarations: [  
      AppComponent  
    ],  
  });
```

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

afterAll() method

The **afterAll()** method provides a function you can define to be run upon completion of all the unit tests within the **describe()** method. Typically, this could clean up any setup work done in the **beforeAll()** method.

afterEach() method

The **afterEach()** method takes a function to perform after each test has been completed. You'd typically need this method to clear any dependencies that your **beforeEach()** call might have set up.

Karma test runner

The Karma test runner is the application that is run via the **ng test** command. It opens a browser and runs the tests from within the browser window. For example, running our test example shows the following output in a browser window.

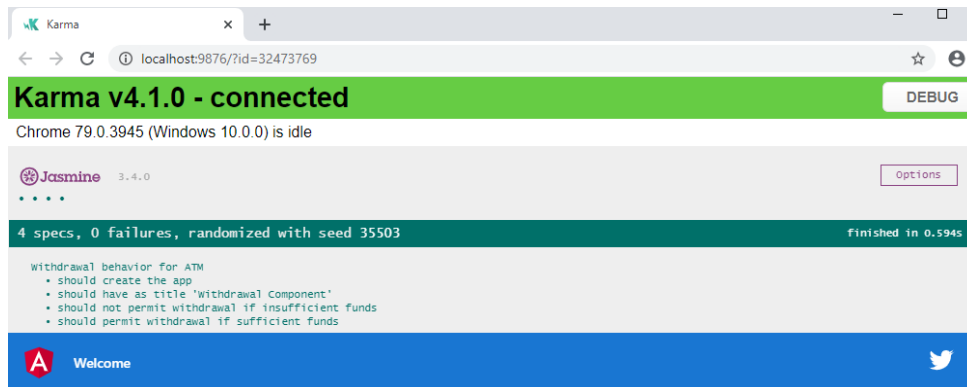


Figure 18: Karma window

The window will show you the browser version you are running, the results of the tests, and the actual output from your component.

Configuration

The Karma application looks for configuration information in a series of file names. Angular CLI generates a **karma.conf.js** file (first file name searched for). The configuration file contains a **module.exports** method that points to a function that sets configuration options. The function receives a single argument, the configuration object. Code Listing 24 shows a partial configuration function.

Code Listing 24: Configuration function

```
module.exports = function (config) {  
  config.set({  
    basePath: '',  
    port: 9876,  
    colors: true,  
    logLevel: config.LOG_INFO,  
    autoWatch: true,  
    browsers: ['Chrome'],  
    singleRun: false,  
    restartOnFileChange: true,  
    frameworks: ['jasmine', '@angular-devkit/build-angular'],  
  });  
}
```

You can find an explanation of the parameters at [this website](#). You might need to change the version number (4.0) if you are using an older version, or if a new version comes out.

Installing additional browsers

By default, Angular creates the Karma configuration file to use the Chrome browser. You can configure it to use additional browsers as well. To use a new browser, you need to install a plug-in to launch the browser. For example, the following command will install the Firefox browser plug-in.

```
npm install karma-firefox-launcher --save-dev
```

Once the plug-in is installed, you also need to add it to the configuration file.

```
plugins: [  
  require('karma-jasmine'),  
  require('karma-chrome-launcher'),  
  require('karma-firefox-launcher'),  
]
```

These are the configuration changes to add and install the browser plug-ins. You can find a complete list of browser plug-ins [here](#).

Adding browsers to the test

Once installed, you can update the browser array in the configuration file to include each browser you've installed and want to run your test on. For example, the following code snippet will allow you to run your unit tests in both Chrome and Firefox.

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

When you execute the test runner, each browser will be open simultaneously, and the tests will be run. If you target multiple browsers with your application, this allows you to ensure your application runs properly in the browsers.

Code coverage

Code coverage is a simple percentage metric to indicate how much of your source code is tested by your unit tests. The Karma test runner can generate code coverage information as part of running your unit tests.

To generate the test coverage report, you still need to run `ng test`, and pass the argument `code-coverage`:

```
<project folder> ng test --code-coverage
```

After the tests are complete, a folder called **coverage** will be generated. This folder contains a project folder with an HTML report (index.html) showing the code coverage. Open the file with a browser, and you will see a report like the one in Figure 19.

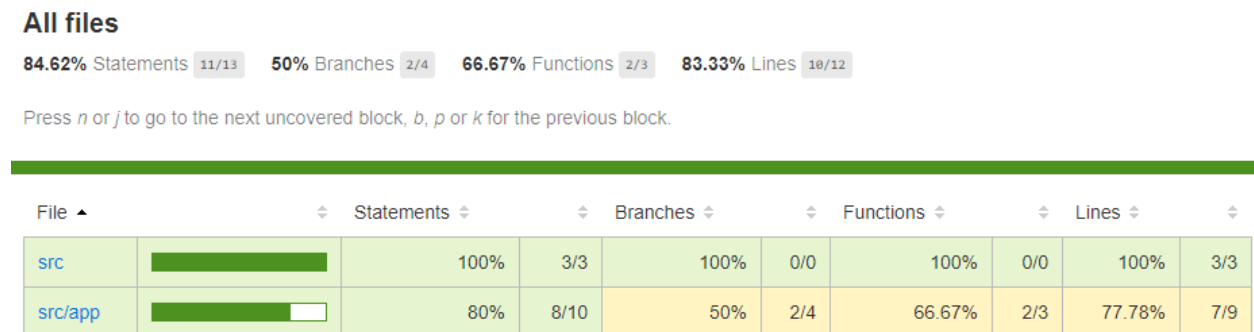


Figure 19: Sample code coverage report

The top row shows the files reviewed and percentage of code coverage. You can drill down to specific folders or even component-level coverage by clicking on a file.

Folder-level view

When viewing the report at the folder level, the report shows summary information for all files within the folder. If I open the **src/app** folder, I might see a report like the one shown in Figure 20.

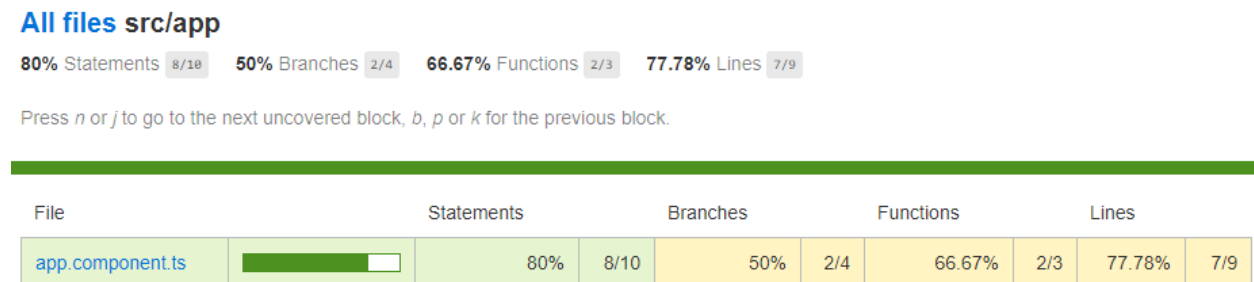


Figure 20: Source folder coverage report

Often, in continuous development environments, a minimum coverage percentage is required to deploy your application.

File-level view

The file-level view provides details as to what code is being covered. Figure 21 shows a sample app component coverage view.

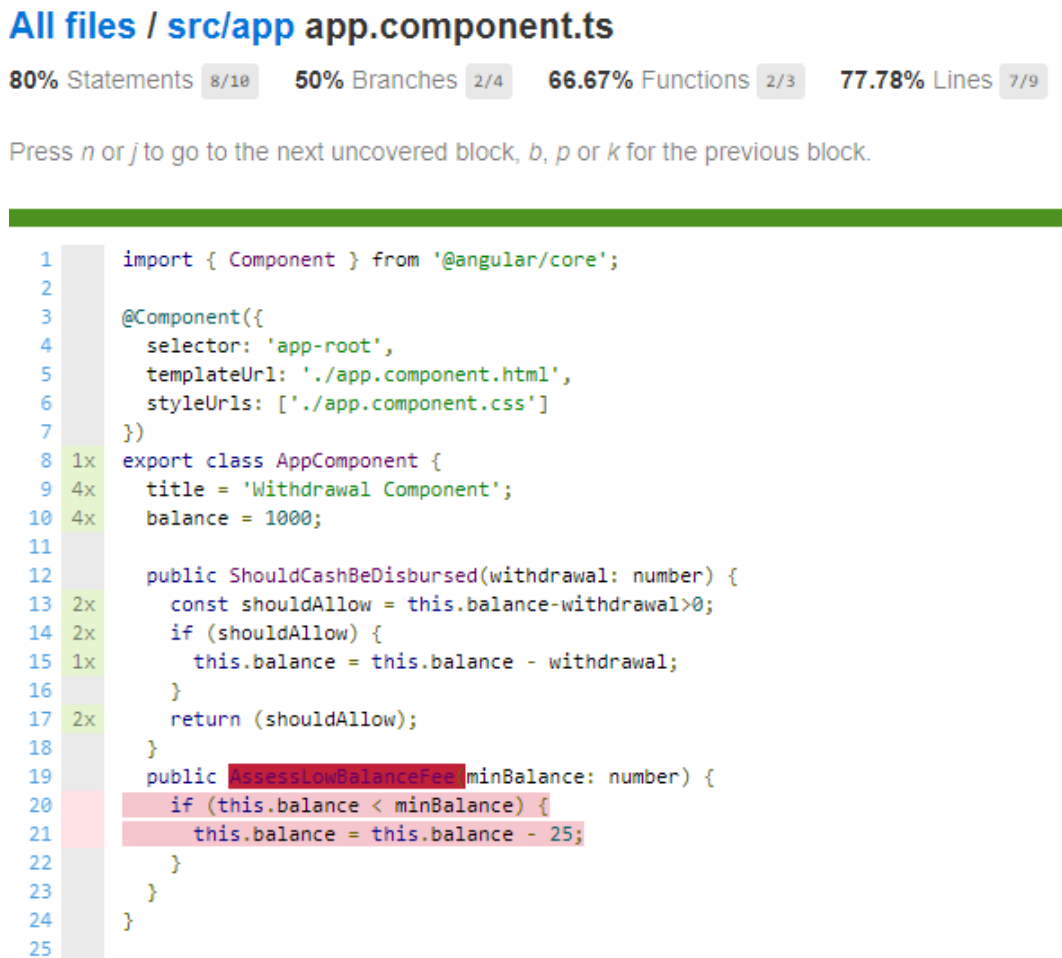


Figure 21: File-level code coverage

The red highlights indicate sections of the code that are not covered by the unit tests. In this example, we have no unit tests for the **AssessLowBalanceFee** method. This is also reflected in the function percentage. 67% of the functions are covered (two out of three, the component itself and the **ShouldCashBeDisbursed** method).

Statement coverage (8/10) reports whether each statement has been covered. The `if` statement, for example, is a single statement, even though it's written in two lines. Line coverage (7/9) reports whether each executable line has been covered. The code in Figure 21 has nine executable lines, but only seven of them have been tested. The green count in the margin indicates how many times the line has been executed. In our test case, we have two calls to the `ShouldCashBeDisbursed` method, but the `shouldAllow` value is true in only one of these; the balance reduction code only is tested once.

The branches coverage metric (2/4) deals with conditional code. There are two conditional statements in this component (the `if` statements). In general, both sides of an `if` statement should be tested. In this simple example, it is probably okay that only the `if` was tested. But if the bank instituted a new business rule, like charging a person \$5 for attempting to withdraw too much money, it would be useful to know we did not test that condition in the code.

Code coverage reports can be very useful to help you and your analysts gain confidence that the unit tests are truly testing your application. If a new developer adds code to charge that \$5 fee, and forgets to add a unit test, the code coverage report will immediately identify the missing test.

Designing your unit tests

When you design your unit tests, there are a few rules and suggestions you should consider for making usable and useful tests.

Naming your tests

The test names should be very descriptive, not `test1` and `test2`. I would much rather know that my test `Balance reduced after withdrawal` failed than `test43` failed. Part of the benefit of Jasmine syntax is that a business analyst can read the `it()` description and know what is being tested. Unless you have psychic business analysts, `test43` isn't a good name.

Test boundaries as well as good input

It is easy to write a test for all good conditions, like when the withdrawal amount is properly compared against the balance. However, you should also test oddities, such as “what happens if the withdrawal amount is a negative number?” Or “what happens if I exceed the available cash in the ATM machine?”

Small tests

If you find yourself writing more than a screen worth of code, your unit test is probably too large. While a unit test isn't necessarily tied to a method, it shouldn't be testing many conditions in a single test. Ask yourself if each new `expect()` method in your test should really be a new test.

Don't persist data

A unit test should not persist any data for other tests to use. One of the options in Karma is to run the tests in random order (always a good idea), so a test may pass if a prior test left needed data around, but will fail if the prior test is not yet run. Always assume each test is independent of other tests.

Summary

The out-of-the-box unit tests provided by Angular give you a solid starting point to improve your development cycle by unit testing. In this chapter, we covered the basics of how to write and run your unit tests, and how to make sure a good portion of your component's code is tested.

While unit testing may seem like a bit of effort to set up, it is of great benefit to ensure your developed code works as expected, and to help identify any bugs quickly, before they reach integration test status. Particular advantages of unit testing include:

- The earlier in the process a bug is found, the less expensive (in terms of time and money) it is to repair. It also makes your debugging process easier.
- Any failed test after a new code check-in can easily be identified.
- Unit testing encourages you to think before coding.

Take the time to add unit testing to your development cycle and enjoy the benefits.

Chapter 8 E2E Testing with Jasmine

Unit tests in the Angular framework are written using the Jasmine library. You can also use the same Jasmine framework commands for writing your end-to-end (E2E) testing code. In this chapter, we will cover how to update E2E configuration files and write E2E tests.

E2E tests rely on a testing library called Protractor, which reads your scenario files and runs them in the browser. When your test files are written in Jasmine or Cucumber (covered in the [next chapter](#)), Protractor is the library to run the actual tests.



Note: Protractor uses Selenium Server to run the actual test code. The Protractor installation includes a helper tool called *webdriver-manager* to get the server running.

Protractor will be installed as part of a project created by the CLI. You can also manually install it using the following command:

```
npm install -g protractor
```

You can run the following command to check that Protractor is installed properly, and is the latest version (5.x as of this writing):

```
protractor -version
```

Configuration

Protractor must be configured with several options, including the specification files you want to run, the **baseUrl** (typically `localhost:4200`) for your application, and the framework name (**jasmine**). Code Listing 25 is a sample configuration file for Protractor generated by the Angular CLI.

Code Listing 25: *protractor.config.js*

```
exports.config = {specs: [
  './src/**/*.e2e-spec.ts'
],
capabilities: {
  browserName: 'chrome'
},
directConnect: true,
baseUrl: 'http://localhost:4200/',
framework: 'jasmine',
};
```


Although Protractor can work with different drivers for the browser, we will cover the direct connection option, which works with Firefox and Chrome. This is the default setup for the E2E folder created by the Angular CLI.

Within an Angular project, there will be an E2E folder, which contains **Protractor.config.js** and a **tsConfig.json** file. The **tsConfig.json** file extends the **tsConfig** file from the parent folder and tweaks a few options for use when Protractor is running. Code Listing 26 shows the default **tsConfig** file.

Code Listing 26: Default E2E tsConfig.json file

```
{
  "extends": "../tsconfig.json",
  "compilerOptions": {
    "outDir": "../out-tsc/e2e",
    "module": "commonjs",
    "target": "es5",
    "types": [
      "jasmine",
      "jasminewd2",
      "node"
    ]
  }
}
```

All the file does is adjust the compiler options and output folder to allow E2E to run.

Protractor.config.js

This file is the primary source of settings for Protractor to run. The file exports a configuration object with settings necessary for Protractor. You can view all available settings [here](#).

Defining browsers

Each browser you want to test in is included in the capabilities section of the configuration file. If you're only testing one browser, the section will contain an object. If you're testing multiple browsers, the section name must be **multiCapabilities**, and should be an array of objects, one per desired browser. Code Listing 27 shows a sample configuration to run both Chrome and Firefox browsers.

Code Listing 27: multiCapabilities section

```
multiCapabilities: [  {  browserName: 'chrome' },
                      {  browserName: 'firefox' }
                    ],
```

Note that you might need to update your webdriver manager to make sure alternate browsers work properly. You can do this by running the following command in your project folder:

```
node node_modules\protractor\bin\webdriver-manager update
```

Now, when you run the E2E command, the tests will be run in all browsers specified.

Firefox

Firefox cannot access the browser logs, so the function **afterEach()** in the spec files needs to check for the browser name before attempting to read the logs. Code Listing 28 shows the revised **afterEach()** function.

Code Listing 28: Revised afterEach function

```
if (browserName !== 'firefox') {
  const logs = await browser.manage().logs().get(logging.Type.BROWSER);
  expect(logs).not.toContain(jasmine.objectContaining({
    level: logging.Level.SEVERE,
  } as logging.Entry));
}
```

You will need to get the current browser name during the **beforeAll()** function. Code Listing 29 shows the **beforeAll()** function.

Code Listing 29: beforeAll function

```
beforeAll(() =>{
  browser.driver.getCapabilities().then(function(caps){
    browserName = caps.get('browserName');
  });
})
```

While it might not be necessary to know the browser name in all instances, this is a good example of using the **beforeAll()** method to define some variables you might need in your specification files.

Jasmine options

The **JasmineNodeOpts** section lets you set options to configure how Jasmine will run. Table 1 shows a list of the possible Jasmine options.

Table 1: Jasmine options

Option	Description
showColors	If true, print colors to terminal.
defaultTimeoutInterval	Timeout in milliseconds before a test will fail.
print	Function to call to print Jasmine results.
grep	Grep string to filter specification files to run.
invertGrep	If true, inverts the grep to exclude files.
random	If true, run the tests in random order.
seed	String to set seed to if using random.

The Angular-CLI-generated file sets the **showColors** and **defaultTimeoutInterval** values, and generates an empty **print** function.

Test files

Once you've configured your Protractor setup (or used the Angular-generated one), you need to explore the actual specification test files that will be used to run the E2E tests. These can be found in the **src** directory of the **e2e** folder. You will generally find two files, one called **app.po.ts**, which declares the application page object (called **AppPage**), and the actual spec file, **app.e2e-spec.ts**, which contains tests to run.

App.po.ts

This TypeScript file will define the application page and provide methods for common actions, such as navigating to a URL, getting elements from the screen, and interacting with elements.

Code Listing 30 shows the default Angular-CLI-generated **appPage** class.

Code Listing 30: Default **app.po.ts**

```
import { browser, by, element } from 'protractor';
export class AppPage {
  navigateTo() {
    return browser.get(browser.baseUrl) as Promise<any>;
  }
}
```

```

    }
    getTitleText() {
        return element(by.css('app-
root .content span')).getText() as Promise<string>;
    }
};

```

This file includes some modules from the Protractor API (**browser**, **by**, and **element**) and provides a couple of method calls to navigate (using the **browser** object) and search for an element (using the **by** object) using some of the element's properties.

Typically, you will add additional methods to this file to simplify and provide common methods for your specification files.

Protractor API

To write your E2E tests, you will be using the objects from the Protractor API. This API allows you to access the browsers, and to grab elements from the browser screen. You can get the text of the element, click an element, send text to the element, and so on. All the operations your user can do manually are available to the API.

browser()

The **browser** object provides commands to control the browser running the E2E tests. It is browser agnostic, so the commands should work for all browsers you configure your tests to run on. The Protractor **browser** class inherits from the **webdriver-manager** class and adds methods and properties for E2E testing. You can find the complete documentation for the **browser** class [here](#).

Navigation

The **browser** object provides a **get()** command, which is used to navigate to a URL. For example, Code Listing 31 provides example code (which could be added to the **AppPage** object) that allows you to navigate to the home (**baseUrl**), a page within the application, or an external URL.

Code Listing 31: Navigation methods

```

navigateTo() {
    return browser.get(browser.baseUrl) as Promise<any>;
}
navigateToPage(url: string ) {
    return browser.get(browser.baseUrl+'/'+url) as Promise<any>;
}

```

```
navigateToUrl(url: string ) {  
    return browser.get(url) as Promise<any>;  
}
```

actions()

actions() is a method that allows you to chain together steps to perform in the browser. For example, you might want to test moving the mouse up or down or test a drag-and-drop operation. You build a chain of action methods, but they are not executed until the **perform()** action is executed.

The following is a list of actions that can be chained together:

- **click()**
- **doubleClick()**
- **dragAndDrop()**
- **keyDown()**
- **keyUp()**
- **mouseDown()**
- **mouseMove()**
- **mouseUp()**
- **sendKeys()**

You create a chain of actions, then add the **perform()** action to execute the defined list of steps. If you want to perform a drag-and-drop test, you will need to find two elements using the **element** methods.

Code Listing 32 shows example code to show a drag-and-drop operation in your test file.

Code Listing 32: DragAndDrop example code

```
let tgt = element(by.id("Amt"));  
let src = element(by.id("Fifty"));  
browser.actions().dragAndDrop(src,tgt);
```

This code grabs the element value using the **id** of “**Fifty**” and drops its value to the input element using the **id** of “**Amt**”. We will cover the **element()** and **by()** methods later in this chapter.

TouchActions()

TouchActions() are like the **actions** method, but allow the chaining of actions related to touch, such as **tap** and **longPress**. The actions are chained together, and when a **perform()** action is reached, the actions are executed. The following touch actions are allowed:

- **doubleTap()**
- **flick()**
- **flickElement()**
- **longPress()**

- `move()`
- `release()`
- `scroll()`
- `scrollFromElement()`
- `tap()`
- `tapAndHold()`

Most of the action methods (either regular or touch) will take an element as a parameter. Once the chain is built, the `perform()` method is added to execute the sequence.

Getting browser information

There are several functions that allow you to retrieve information from the browser, such as capabilities, geolocation, and device time. A few of them are described here. Each method returns a promise, so you need to attach an event handler to receive the result. Code Listing 33 shows example code to capture a promise (the `then()` function).

getCapabilities()

This method returns an object describing the capabilities of the browser. Code Listing 33 shows how to call the method to gather some information about the browser. Since this information will stay the same during execution of the test, you should consider adding it to the `beforeAll()` method in the step definition.

Code Listing 33: Browser capabilities

```
beforeAll( async () =>{
    browser.getCapabilities().then((c) => {
        hasTouch = c.get('hasTouchScreen');
        browserName = c.get('browserName');
        platform = c.get('platform');
    });
});
```

getCurrentURL()

This method returns a promise of a string value holding the current URL. You'll often need this to test expected navigation results.

getGeoLocation()

This method returns an object with the latitude, longitude, and altitude of the device location. There is a corresponding `setGeoLocation()` method that allows you to set locations, in case you want to test location-specific features of your application.

getScreenOrientation()

This method returns a string with either **LANDSCAPE** or **PORTRAIT**, depending upon the device orientation. The `setScreenOrientation()` method lets you set the orientation mode to test scenarios where the application behaves differently based on orientation.

getTitle()

This method will return the browser title for the current window.

Browser behavior

There are also several browser methods that let you change settings on the browser, primarily for testing different statuses, such as Airplane Mode or no Wi-Fi. These can be particularly useful when creating a test plan for mobile applications.

close()

This method closes the current browser window.

toggleAirplaneMode()

This method allows you to toggle Airplane Mode (disable Wi-Fi, Bluetooth, GPS, and so on). You might want to test local usage of your application with Airplane Mode turned on.

toggleWifi()

This method toggles the Wi-Fi setting, allowing you to test behavior as if Wi-Fi is turned off.

sleep()

The **sleep()** method takes a parameter of the number of milliseconds the browser should sleep for. You can use it to allow some time for dynamic code to populate the controls on a form or to simulate user behavior.

element()

The **element** object represents a single element from the screen. It might be some input text, an Angular binding, or a button. In order to get the element, you need to use a locator, which is a method call that provides different ways to find an element. The simplest and most direct locator is **by.id()**, which lets you get an element associated with the element's **id** property. The parameter to the **element()** method must be a locator, and is expected to return a single element (**element.all()** expects a collection of elements).

Locators begin with the keyword **by**, followed by a method name, such as **id()** or **buttonText()**. You can find a complete list of locators [here](#).

Some of the more common locators are listed in Table 2.

Table 2: Common locators

Locator	Description
binding(string)	Returns element with interpolated string. <code>{{ GPA }} </code> <code>by.binding('GPA')</code> would return the <code></code> element.
id(elementId)	Returns the element with the associated <code>id</code> property. <code><input id='pwd'></code> <code>by.id('pwd')</code> would return the <code>input</code> element
buttonText(text)	Returns button element whose text matches the parameter.
model(name)	Returns element associated with <code>ng-model</code> name. <code><input type="text" ng-model="person.name"></code> <code>by.model('person.name')</code> returns the <code>input</code> element.
name(element name)	Returns element with the given name property. Use carefully, since multiple elements can have the same name, in which case the first one is returned, and a warning is given.
css(css selector)	Finds an element by a CSS selector, such as <code>H1</code> (header 1 element) or <code>.Menu</code> element with the class name of <code>Menu</code> .

You can use `$()` as shorthand for finding elements by CSS.

element.all()

`element.all()` works similarly to `element()`, but is expected to return a collection of elements (even if only one element is found, a collection will still be returned). `element.all()` still requires a selector, although the selector should be a scope larger than `id` or `binding`. You might use this method to determine the number of items in a list box, or menu options. Code Listing 34 shows an example of using `element.all` to retrieve all `li` tags from a class called `menu`.

Code Listing 34: Retrieve menu items

```
let menus = element.all(by.css('.menu li'));
```

Once you have the variable, which is a collection of elements, you can use the methods shown in Table 3 to get details of or manipulate the collection.

Table 3: *Element.all methods*

Method	Description
<code>get()</code>	Gets an element by the index parameter (zero-based).
<code>count()</code>	Returns number of elements in the collection.
<code>first()</code>	Gets the first element from the collection.
<code>last()</code>	Gets the last element in the collection.
<code>each()</code>	Performs a function on each element in the collection.
<code>map()</code>	Applies a map function to the collection. The function gets the element and index as parameters, and returns a collection of the object built by the map.

You can use `$$()` as shorthand for `element.all(by.css())`.

Working with elements

Once you've identified an element, you can interact with it to send text, click a button, check value, and so on. You will typically save the found element to a variable, and then process commands against the element.

The following are the common methods on the element.

isPresent()

This method is used to determine whether an element exists on the page. For example, the following code snippet checks to see if an element with the `id` of `SaveBtn` is on the current page.

```
element(by.id('SaveBtn')).isPresent()
```

isEnabled()

This method will test whether the element is currently enabled. For example, in some applications, a Save button stays enabled until all errors are cleared. This would allow you to confirm that behavior for applications designed using that approach.

isSelected()

This method indicates whether an element is selected or not. The following code snippet shows how we might determine whether taxes should be applied in an e-commerce application.

```
<input id="ApplyTaxes" type="checkbox">
```

```
var taxes = element(by.id('ApplyTaxes'));
taxes.click();
expect(taxes.isSelected()).toBe(true);
```

isDisplayed()

This method is used to determine whether the current element is displayed on the screen. Note that if you use the Angular `*ngIf` directive to control an element, the element is either added to the DOM, or not. The `isDisplayed()` method should not be used to test `*ngIf` conditional elements; use `isPresent()` instead.

getText()

The `getText()` method returns the value of the element without any leading or trailing spaces. It returns the content of the HTML element's `innerText` property. Note that the element must be visible for this method to return the value.

clear()

The `clear()` method sets the value of the element to an empty string.

click()

This method clicks the element, either by selecting the element (such as an input field) or performing the associated action (such as with buttons or links).

submit()

This method should be on a form element and will perform the submit action associated with the form.

sendKeys()

This method allows you to “type” text characters into an editable HTML element. Note that the text will be appended, so you’ll need to clear the previous text if you want new text in the element. The following code snippet shows an example of the method.

```
enterNewText(id:string, text:string) {
    let elt = element(by.id(id));
    elt.click();
    elt.clear();
    elt.sendKeys(text);
}
```

This example clicks in the element, clears the previous text, and adds the new text value to the control.

getCssValue()

This method returns the computed style for an element and property. The method expects a CSS property value as a string parameter. It will return the computed style for the element, using the CSS priority logic to determine the style. For example, you might want to use this method to determine an element's color, expecting red (**#FF0000**) when an error condition has occurred. The following snippet checks to make sure the error message is displayed in red text.

```
<div id="ErrorMsg">Transaction failed</div>
var errMsg = element(by.id('ErrorMsg'));
expect(errMsg.getCssValue('color')).toBe('#FF0000');
```

Note that colors will be returned as hex values, regardless of how the color was computed.

getAttribute()

This method returns the current value for an attribute on the element. Attributes that are Boolean (such as **autofocus**, **checked**, and **selected**) will be returned as either true (attribute is set on the element) or null (the attribute is not on the element). The **class** attribute returns a string value of the element's current class. The **style** attribute returns a representation of the current style settings as a delimited string.

getTagName()

This method returns the HTML tag that the element was found within. For example, if we wrapped the error message in a **** tag, the method would return the text **span**.

App.e2e-specs.ts

This is the primary file where your specifications are written. These should typically be written so that a non-developer can understand them. The clearer your method names in **po.ts** are, the clearer the spec will read.

We discussed Jasmine a bit in Chapter 6 in terms of unit testing. Jasmine is also used for E2E testing. Combined with the Protractor methods to control the browser, we can mimic the behavior of a user and define expected outcomes.

describe()

Typically, you want to tie the E2E test to a user story (an Agile framework artifact describing the functionality from a user point of view). Your spec files will begin with a **describe** function, generally some test that indicates which user story you are testing. In our banking example, we might have a user story as shown in Code Listing 35.

```
As a "Bank Customer"  
I want "To withdraw cash"  
So that "I can have a fun night out"
```

With this story, we can create a **describe** function such as:

```
describe("Withdraw cash from ATM", () => {} )
```

It is possible to create a single spec file for your entire application, but I would recommend keeping the spec files small and focused on one part of the application.



Note: Job stories are an alternate to user stories, focusing on an event, such as “when reviewing order history.” In this case, the user reviewing the history is not as important as the action being done. The format of the job story is: When <situation> I want to <action> so I can <expected outcome>. Whatever story format your analysts use, be sure it is clear in your test specs which story the test is related to.

it()

The **it()** method is where you will describe each part of the story being tested. You will use the various Protractor **browser** and **element** methods to make the browser mimic the behavior you’d expect the user to perform. It takes a string (description test name) and a function to perform the test. The description name should provide a person reading the spec a clear explanation of what part of the user story is being tested. For example, for the “withdraw cash from ATM” story, you might have test methods like:

```
it("Allow the user to withdraw cash")  
it("Display an error for insufficient funds")  
it("Display an error if trying to withdraw more than daily limit:")
```

Keep in the mind that the test spec file is also documentation for the business analyst, so even though they might not understand the code, using good descriptions and methods names will go a long way.

Structure

The generated spec file, and in general, the ones you write as well, will follow the structure of the code. The script should consist of **describe** and an anonymous function:

```
describe("User story description", () => {  
  
}
```

The first part of the function should declare the variables to be used through the script. At minimum, a variable will be created from the class defined in the **po.ts** file. Any additional variables needed should be defined as well. Code Listing 36 shows a simple variable declaration list and a **beforeAll** method (called once before the tests start) to populate some variables.

Code Listing 36: Partial spec file (startup)

```
describe('Withdraw cash from ATM', () => {
  let page: AppPage;
  let browserName: string;
  let hasTouch: boolean;
  let platform: string;
  let BankBalance: number;
  beforeAll( async () =>{
    browser.getCapabilities().then((c) => {
      hasTouch = c.get('hasTouchScreen');
      browserName = c.get('browserName');
      platform = c.get('platform');
    });
  });
});
```

You can initialize other variables the test might need. In our example, we are creating the object and declaring a bank balance variable. We also update a few browser properties we might use in the test.

Code Listing 37 shows the **beforeEach** method, which is called prior to each individual test being run. For each test, we create a new copy of the **AppPage** object and reset the bank balance to \$1,000.

Code Listing 37: beforeEach method

```
beforeEach(() => {
  page = new AppPage();
  bankBalance = 1000;
});
```

Once we've set up our variables and before methods, we can write our actual tests using the Protractor methods to run the test. Keep in mind that the **po.ts** file should provide some wrapper functions to make the code clearer. For example, the following code snippet will find an element and enter text into it.

```
let elt = element(by.id(id));
elt.click();
elt.clear();
elt.sendKeys(text);
```

However, to assist the business analysts, I've added a wrapper function around those commands called `enterText(id, text)`. This is a bit more readable to the spec reader.

```
page.enterText("amount", "250"); // Enter $250
```

In a similar fashion, my `po.ts` file contains definitions for clicking a button and menu link, as shown in Code Listing 38.

Code Listing 38: po.ts wrapper functions

```
enterText(id:string, text:string) {
    let elt = element(by.id(id));
    elt.click();
    elt.clear();
    elt.sendKeys(text);
}
clickMenuLink(id: string) {
    return element(by.id(id)).click();
}
clickButton(id: string) {
    return element(by.id(id)).click();
};
async resultMsg() {
    let elt = element(by.id('resultMsg'));
    return elt.getText();
};
```

Even though the `clickMenuLink` and `clickButton` methods contain identical code, by using different method names, we allow the end user to read the definition in the test steps and get a better understanding of what the test is doing (such as understanding if this step is clicking on a menu or a button). Code Listing 39 shows the test to withdraw money.

Code Listing 39: Withdraw money example

```
it('Should allow me to withdraw $250 from checking', async () => {
    page.clickMenuLink("withdraw"); // Go to withdrawal page
    page.enterText("amount", "250"); // Enter $250
    page.clickButton("withdrawMoney");
});
```

While the method shows the action, we still need to see if the test worked. We call our `withdraw` component method to get a Boolean true or false value back. Code Listing 40 shows the completed method.

Code Listing 40: Completed withdraw \$250 test

```
it('Should allow me to withdraw $250', async () => {
    page.clickMenuLink("withdraw");           // Go to withdrawal page
    page.enterText("amount", "250");          // Enter $250
    page.clickButton("withdrawMoney");
    expect(page.resultMsg()).toBe('OK');
});
```

Once the sequence of steps is performed, **expect()** is evaluated to determine whether the test passed. In this case, expect the status message to be OK, since the withdrawal was allowed.

Code Listing 41 shows the failed withdrawal case, attempting to withdraw \$1,250 instead of \$250.

Code Listing 41: Failed withdrawal test

```
it('Should not allow me to withdraw $1250', async () => {
    page.clickMenuLink("withdraw");           // Go to withdrawal page
    page.enterText("amount", "1250");          // Enter $1250
    page.clickButton("withdrawMoney");
    expect(page.resultMsg()).toBe("Insufficient funds");
});
```

Keep your test simple and readable. The more you hide the programming complexity inside the **po.ts** file, the easier the spec file should be for business analysts to read.

afterEach method

The **afterEach()** method is called upon completion of each test (**it()** method) in the spec file. This can be useful for any cleanup work, or checking the browser logs for errors, like the default code from the Angular CLI does. Code Listing 42 is the **afterEach()** method from the spec.

Code Listing 42: afterEach method

```
afterEach(async () => {
    if (browserName !== 'firefox') {
        const logs = await browser.manage().logs().get(logging.Type.BROWSER);
        expect(logs).not.toContain(jasmine.objectContaining({
            level: logging.Level.SEVERE,
        } as logging.Entry));
    }
});
```

Note that the Firefox browser does not allow access to the logs, so by using the browser name variable we populated during the `beforeAll()` method, we check to only review the logs for errors if we are not using Firefox.

Summary

E2E testing, while not a final test (QA departments are still needed), provide a good sense that the overall application flow is working. By creating good method names inside your `po.ts` file, you can produce a readable specification file that can also drive your tests. While it takes a bit of effort to set up, and can take some time to run, it is a time-saver for an overall testing plan. This allows QA to focus on subtle bugs, or application bugs where the specification doesn't quite do what is expected. And whenever QA does find a bug, you can add it to your E2E tests to be sure it is caught the next test run, before QA gets their hands on it.

Chapter 9 E2E Testing with Cucumber

While Angular CLI will set up your E2E testing to use Jasmine, the Cucumber testing framework can also be used for end-to-end testing. The Jasmine framework allows flexibility in how the tests are named, simply as strings in the `describe()` and `it()` methods. Using the Gherkin language and Cucumber allows a bit more structure to the way the feature definition files are written.

Cucumber consists of two primary file types. The feature file describes the tests in a descriptive language and can generally be written by business analysts. This allows a non-developer to describe what the code must do. Listing 43 shows a simple example feature file.

Code Listing 43: Feature file example

```
Feature: Verify Authentication
  As a user,
  If I am not authenticated, I should be navigated to initial screen
  After successful authentication, I should be allowed to access Application

  Scenario: When I click Login, I am navigated to login page
    Given I am on the Initial Screen
    When I click on Login
    Then I am rerouted to Login page

  Scenario: On Login page, I enter an invalid username and password
    Given I am on the Login Screen
    When I enter username "BadEmail@jbc.com" and password "badPassword"
    Then Invalid credentials message displayed

  Scenario: On Login page, I enter a valid username and password
    Given I am on the Login Screen
    When I enter username "GoodEmail@jbc.com" and password "pass"
    Then I am navigated to landing page
```

In addition to the feature files are the step definition files. These files consist of a pattern (that matches one or more entries in the feature file) and the snippet of code necessary to test that feature. These are typically written by the developer to translate the business requirements defined in the feature file into test conditions and results.

This combination creates a living spec document. If the analyst changes the feature file, the tests should break, which means the underlying code needs to be reviewed.

Feature files

Feature files are specification files for defining acceptance tests. They are based on the Gherkin language, which is a simple language designed to be created and read by business analysts. Due to its simple syntax, the Cucumber testing framework can read this language to run the acceptance tests.

The general format of the language is a keyword, followed by some text, such as:

Given I enter a bad password

Then I see a bad credentials message in the error window

Some keywords are primarily for documentation purposes, describing what is being tested in the file. The **Given**, **When**, and **Then** keywords typically perform actions and test for results. The Gherkin keywords are described in this section.

Feature

The feature is a high-level overview of what is being tested, such as the login page or a shopping cart. The feature description begins with the keyword **Feature:**. All feature files must begin with the keyword feature, followed by a colon.

After the keyword is free-form text, which ends when a keyword is found. Typically, the feature can be a story or some other description of what this file will test. The file itself will have multiple examples detailing what should be tested to confirm this feature works.

A user story typically has the format:

As a <role>, I want <some goal> so that <some reason>

A user story is typically small enough that a team can complete it within a single sprint. Another story format is the job story, which has three parts, but a different view:

When <situation>, I want to <some task> so I can <expected outcome>

Job stories focus on something occurring without a specific role, while user stories focus on the person and why they want to perform the task. With either story approach, or even just descriptive text, you should keep the content consistent so anyone opening the feature file can quickly tell exactly what should be tested.

The feature being tested will have multiple examples or scenarios that, taken as a whole, should completely test the feature.

Example

The **example** (or scenario) keyword is one small test of the feature file. These are the actual tests that will be run, and should handle both positive results (expected behavior) and error conditions (such as bad data being entered). In general, the test describes a starting point (application page, for example), action taken (entering a good password), and expected result. By using the Gherkin syntax, a feature file and examples should be readable, so business analysts can write what they expect the example to test out. The scenario will consist of the Gherkin keywords to describe preconditions (**Given**), actions taken (**When**), and expected outcomes (**Then**).

Given

This command specifies a precondition that needs to occur before the scenario actions are taken. For example, we might start the test with a statement such as:

Given “I am on the splash screen”

An entry in the step’s definition will translate this into the necessary browser navigation to get to the application’s initial screen.

When

This command represents the action a user might take, such as clicking a button or entering text. Our example in [Code Listing 43](#) combines **Given** (precondition) and **When** (action taken) to define what the user is doing. Although with end-to-end testing, the system will perform the steps, a quality assurance resource could manually read the feature file and perform the test steps manually to confirm the feature works.

Then

The **Then** command is used to describe the expected outcome of the test example. In our example, some actions lead to another webpage, while others display an error message. The steps definition will provide the proper comparison code to confirm the test worked.

And/But

The **And** and **But** commands are primarily used to improve the readability of the feature file. They basically continue the previous command. For example, if you had two preconditions:

Given “I have successfully logged in”

Given “I am on the main application screen”

To make the feature file more readable, you could express these as:

Given “I have successfully logged in”

And “I am on the main application screen”

Similarly, expected outcomes such as:

Then “I should see the grid results”

Then “I should not see an error message”

Could be written as:

Then “I should see the grid results”

But “I should not see an error message”

The **And** and **But** commands rely on the previous keyword to interpret their actual meaning.

Background

The background keyword (**Background:**) is used to specify some common steps that should be used with each scenario. The background section of commands (typically **Given** and **When** commands) is run prior to each scenario. This eliminates the need to repeat code multiple times in the feature file.

Scenario outline

You might create scenarios where the steps are the same and need to be repeated for a set of values. This is what the **Scenario Outline** (or scenario template) keyword allows you to do.

Let’s imagine you want to ensure that an error message matches the user-selected language. You might have the following scenarios defined:

Given I’ve set the language to French

When An invalid password is provided

Then The message should be Mot de passe incorrect

Given I’ve set the language to Spanish

When An invalid password is provided

Then The message should be Contraseña invalida

As more languages are offered, these scenarios can become inefficient and time-consuming to enter each time.

Using the scenario outline feature, we can define a table and create a single scenario that would be able to call values from the table.

Scenario Outline

Given I've set the language to <language>

When An invalid password is provided

Then The message should be <passwordMessage>

Examples:

Language	passwordMessage	
French	Mot de passe incorrect	
Spanish	Contraseña invalida	
English	Invalid password	

The scenario will be repeated three times, one for each row in the examples table. The first row is the column headers and provides the names of the variables to substitute in the scenarios. The table values will be tested one row at a time.

If you find yourself writing scenarios that are the same thing with slight variations, consider the **Scenario Outline** keyword and a table of values to loop through. You can use the keywords **examples** or **scenarios** to begin the table definition. The table can have as many rows and columns as needed for your test.

Parameters

The text following the action commands (**Given**, **When**, **And**, **But**, and **Then**) is typically a line of text. You can include parameters within that line by putting the parameter in quotes. This simplifies the coding required in the step definition files. For example, consider the following **Then** lines:

Then I see a bad credentials message in the error window

Then I see a logged-in message in the success window

These two lines would require separate steps since the text is different. However, by using parameters, the lines would now look like this:

Then I see a "bad credentials" message in the "error" window

Then I see a "logged-in successfully" message in the "success" window

Readability doesn't suffer much, and it allows a single step to be used to handle either condition. We will cover this in more detail when we review the step files.

Example feature file

Let's take our example feature file from the beginning of this chapter ([Code Listing 43](#)) and expand it to complete the scenario. The first action is to break this into two feature files: one to test the initial screen (copyright notice appears, help button appears, etc.), and a second to test the login screen. Feature files should be small, testing small areas of the application. Although there is no hard and fast rule, it is general practice that a feature file covers a single application page.

Structure

Typically, you will have the following folders with the E2E folder structure:

- **Features:** Feature files written by business analysts.
- **Src:** Common page object files to define **AppPage** and other classes.
- **Steps:** Code to implement features.

You can customize your setup any way you'd prefer, however, by updating the **Protractor.conf** file to provide folders for features and steps. The **src** file (page objects) are referenced directly in the step definitions.

Steps

The Steps folder should contain the code to implement the statements found in the feature files. A step definition will contain a keyword function (**Given**, **Then**, or **When**) followed by a text string, and a function to implement the step.

During the testing, all the specified step files are combined into a collection of step functions. As the feature files are processed, the feature lines are looked up across the entire set of step files. This can cause some unexpected behavior if a step definition that resolves the feature is found in an unexpected file. In addition, if multiple steps are found that could match the feature line, it will be flagged as an ambiguous step definition as shown:

Given I am unauthenticated user

Multiple step definitions match:

I am unauthenticated user - e2e\steps\common.steps.ts:750

I am unauthenticated user - e2e\steps\authentication.steps.ts:14

In this case, you need to change the wording in the step definition to remove the ambiguity. As a matter of practice, I suggest creating a **common.step.ts** file, which will contain any steps that are applicable to any page in the application, such as navigation between pages or clicking buttons. For each feature file that **might** need specific step definitions, create a corresponding step file to implement just those steps.

Page object files

The page object files (**app.po.ts**) typically contain some Protractor methods to interact with the webpage. The default page generated by Angular is the **app.po.ts** file, which defines the **AppPage** class. This class has some basic methods likely to be used throughout the application test. If you need specific features added to test an application page, you can extend this class to create a new class for that page. The following snippet shows an example:

```
export class WithdrawalPage extends AppPage
```

Common methods are added to this file, so all definitions can use these commands.

Navigation

Most feature files will start at a page within your application. Code Listing 44 shows code to navigate to the home page (base URL) and a page within the application.

Code Listing 44: Navigation

```
async navigateToHome() {
    return await browser.get(browser.baseUrl) as Promise<any>;
}
async navigateToPage(url: string = '/'): Promise<any> {
    url = url[0] === '/' ? url : config.baseUrl + url;
    if (url.length > 1) {
        url = url[0] + url[1].toUpperCase() + url.slice(2);
    }
    return await browser.get(url);
}
```

Clicking elements

Another common task is to click on an element, such as a button or a link.

Code Listing 45: Clicking elements

```
async clickButton(btn: string) {
    browser.driver.findElement(by.id(btn)).click();
}
```

Even though clicking a link would be identical code, I would suggest having a **clickLink(link: string)** method as well to improve readability.

Entering text

You will often need to enter text into an edit control. Code Listing 46 shows a method that takes an element `id` and a string of text and enters the text into the control (first clearing the text).

Code Listing 46: Enter new text

```
async enterText(id: string, text: string) {  
    const inputField = await browser.driver.findElement(by.id(id));  
    await inputField.clear();  
    await inputField.sendKeys(text);  
};
```

Code Listing 47 is a slight variation called `appendText` that performs the same function, but without clearing the previous text.

Code Listing 47: appendText

```
async appendText(id: string, text: string) {  
    const inputField = await browser.driver.findElement(by.id(id));  
    await inputField.sendKeys(text);  
};
```

Selecting an option

If you have a list box control, you might want to select an option from the list by either name or position. To select the element by name, we use the `by.cssContainingText()` locator. This locator takes two arguments: the CSS locator, and the text to find. Code Listing 48 shows a function to click on an element from the list.

Code Listing 48: Click an option in list box by name

```
async selectOption(text: string) {  
    const choice = await browser.driver.  
        findElement(by.idcssContainingText('option', text));  
    choice.click();  
};
```

You can also add a method to select an option by numeric position in the list. Code Listing 49 shows a method called `selectOptionByNumber`.

Code Listing 49: Select option by number

```
Async selectOptionByNumber ( element, Num ) {  
    var options = element.all(by.tagName('option'))
```



```

        .then(function(options){
            options[Num].click();
        });
    };

```

This method relies on `element.all` to return an array of all option tags within the specified select element. It then clicks the array element by number.

Matching steps and features

When a feature file is processed, the keyword and text in the feature file is used to search for a matching step definition (from all step files in the configuration). Our example in Code Listing 50 will match the exact feature “I am on the Access Denied page” when used as part of a **Given** keyword sentence.

However, business analysts might not use that exact wording, so if they enter “I’ve opened the access denied page,” the E2E test will report it and won’t be able to find a matching step definition. Fortunately, you can use regular expressions to help with matching. Code Listing 50 shows a simple step definition for navigation.

Code Listing 50: Access Denied page step definition

```

Given('I am on the Access Denied page', async function() {
    await appPage.navigateTo('access-denied');
    const url = await browser.getCurrentUrl();
    expect(url).to.equal(`${config.baseUrl}/access-denied`);
});

```

If we update the **Given** function to the following, we can match a variety of ways a user might enter the feature to navigate to the Access Denied page. The second parameter (**i**) to the **RegExp()** function tells the expression to be case-insensitive.

Code Listing 51: Access Denied page step definition, with RegExp() function

```

Given(new RegExp("(I am on|I've navigated to|I've opened) the
                  (Access Denied|Denied) (Page|Site|url)$", "i"), async functi
on () {
    await appPage.navigateTo('access-denied');
    const url = await browser.getCurrentUrl();
    expect(url).to.equal(`${config.baseUrl}/access-denied`);
});

```

While the use of regular expressions can help make things easier for the feature file author, it also increases the possibility of more ambiguous step definitions. If another step definition file has the following step, it would be ambiguous with the regular expression version:

Given('I've opened the Access Denied Site')

You need to strike a balance between flexibility for the feature file authors and the complexity of the step definition files. I generally prefer some degree of flexibility, but also providing the feature file authors a template to use for common features.

Using parameters

You can also use parameters in the feature file by enclosing the parameter value in quotes. For example, if we want the user to enter text into a username and password prompt, the feature file statement might be written as:

When I enter "Joe" into the username and "Bad" into the password

The matching step definition would look like:

When('I enter {string} into the username and {string} into the password',

The function would then be written with two parameters, such as:

async function(username: string, password: string)

This would allow the feature file author to use the same syntax when testing a good password for successful login. Code Listing 52 shows a sample step definition to process the login feature line.

Code Listing 52: Sample login step definition

```
When('I enter {string} into the username and {string} into the password',
  async function( username: string, password: string )
{
  const userNameField = await browser.driver.findElement(by.id('UserName'))
);
  const passField = await browser.driver.findElement(by.id('Password'));
  await userNameField.clear();
  await passField.clear();
  await userNameField.sendKeys(username);
  await passField.sendKeys(password);
  await browser.driver.findElement(by.id('Login')).click();
});
```



Note: Parameters and regular expressions cannot be used together.

Our code in Listing 53 directly works with the browser object; however, we could create a simpler version using our methods defined in the page object.

Code Listing 53: Sample login using page object

```
When('I enter {string} into the username and {string} into the password',
    async function( username: string, password: string )
    {
        await appPage.enterText('UserName',username);
        await appPage.enterText('Password',password);
        await appPage.clickBtn('Login');
    });
```

Lookup tables

While there are numerous ways to find an element on the screen, the **by.id** method is the most likely to find a single element (assuming you've added the **id** to the element). However, the feature file authors likely do not know your element **id** values. You can create a lookup dictionary, with a text word and the corresponding **id** for the element. This would allow the feature file to have a statement like:

When I enter 2.50 into the “Tax Amount” field

Your matching step definition would then get the parameter string **“Tax Amount”** and look in the dictionary to discover the field **id** is **TaxAmt**. This allows the feature author to have more flexibility in writing the feature files.

Installation

To install the Cucumber framework so you can use the Gerkhin syntax, you need to run the following installation step:

```
npm install --save-dev @types/{chai,cucumber} chai cucumber
    protractor-cucumber-framework
```

This will allow you to create feature files and step definitions for testing your Angular application. You can visit [this website](#) to look for other implementations.

Configuration

The **protractor.config.js** file will contain the necessary configuration options to indicate the feature files to run and set the Cucumber options to run the tests.

The config file creates a JSON object called **exports.config**. This object holds the various settings needed to run Cucumber tests. Some key properties are:

framework

You need to provide the testing framework to the configuration file, with two options:

```
framework: 'custom',  
  
frameworkPath: require.resolve('protractor-cucumber-framework'),
```

This indicates we are using a custom framework and provides the name for the framework to use.

specs

This property is a list of test specification files. Typically, you will place your feature files in their own directory, and the **specs** property lists that folder and the files to run. The following example runs all feature files in the features folder in the E2E structure.

```
specs: ['./features/**/*.feature']
```

You can have multiple sets in the **specs** property.

cucumberOpts

This section is an object that provides the parameters used by Cucumber to process the tests.

require

This is a list of the various TypeScript files that provide the step definitions for the tests. Assuming you place your step definitions in their own folder, you could use the following:

```
require: ['./steps/**/*.ts']
```

This will load all the step definitions when building the code to match against the features.

tags

The **tags** collection allows you to provide a list of regular expressions that provide control over which feature files are run. For example, you might not want to run any draft features, so you can add the following regular expression to exclude scenarios beginning with the text **@Draft**:

```
tags: [ '~@Draft' ]
```

Business analysts could now add the text **@Draft** prior to the scenario to prevent it from being run or tested while they are still writing the feature.

strict

When this flag is set to true, if any steps are pending or undefined, then the test will fail.

format

The **format** option allows you to specify the location and format of the output report from the test. For example, the following syntax creates a JSON report in the **e2e-report** folder:

format: 'json:.e2e-report/results.json'

Visit [this URL](#) for a complete list of the configuration syntax.

Summary

Cucumber is a testing framework option for Angular that allows analysts to write specifications in a pseudo-natural language, and not worry about the testing details. A developer or QA resource will code-test step definitions to match the specifications found in the feature files. By using a more structured syntax (like Gherkin), you can add some consistency to the test plan and provide a common code base to use to run the tests.

Chapter 10 Planning E2E

While end-to-end testing (E2E) can help create a more robust product and save time by performing common UX tasks, it does take some time investment to set up. In this chapter, we cover some of the steps to consider when first setting up your E2E environment.

What testing framework?

The Jasmine Framework (Angular default) is more free-form. The business analyst simply provides a test description (**it** command) and the QA resource or developer provides the code to perform the test. This allows a free-form test description, which provides flexibility and power to the business analyst, but possibly increases the workload of the developer since there might be descriptions that are the same, but still require code to implement the definition.

The Cucumber approach requires that the business analyst be more structured when writing the test definitions, since they need to provide the prerequisite setup (**Given**), actions (**When**), and expected results (**Then**). By providing structure at the step level, the QA resource or developer can likely develop common code to handle the steps, particularly since many steps are common (such as navigate to a webpage, enter text, or select a value from a drop-down).

Base functionality

Whichever approach you use, be sure to create a code base of the common operations. A person writing the code to perform the test should not need to write code for the following:

- Navigate to a page.
- Clear a text box and enter text.
- Click a button.
- Click a link.
- Select a value from a drop-down.
- Select a radio button or check box.

The focus of the code should be calling methods with different parameters, so the developer is coding the test, not figuring out how to make the browser commands work. Many times, a developer gets involved to come up with a solution to a recurring test step. When this happens, the step should be reviewed for inclusion in a base class.

For example, we had a recent example where the value of a previous drop-down determined what the next drop-down should contain:

Type	Expected
States	List of states and provinces
Rates	List of shipping rates

The solution to resolve “List of...” was to match the expected code to a value that would be found in the list, such as “Pennsylvania” being found in the states list. The code was added to check that a value exists in the list. This is the type of item that should be added to the base class, so future coders facing the “List of...” don’t need to determine the code to confirm an entry exists in a list, but rather can simply call a method like `ListContainsValue()`.

Templates

Computers love consistent data, while humans understand variations that mean the same thing. To a computer, the following commands are two distinct requests:

- Navigate to home page.
- Jump to the home page.

However, to people writing a test spec, the two commands are performing the same action.

One approach is to use regex patterns to provide flexibility in how the business analyst writes the test step. While it takes a bit more work to use a regex pattern as opposed to text, it makes it easier for the business analyst to write the specifications and test plans.

Another approach is to provide templates or snippets with the commonly used test commands. Many editors allow you to define your own code snippets, so creating some for the business analyst’s editor can make it easier for the analyst to write the correct syntax, rather than relying on their memory as to what the actual wording should be.

If you have very common test patterns (such as requiring a “navigate to...”, making sure fields exists, or making sure save and cancel buttons are clicked), then create a template with the proper expected syntax so the spec/test writers use the expected syntax to ensure the test will run.

Summary

E2E testing is a very comforting tool for knowing that the UX is being tested. If manual testing detects an error, you can add the condition to your E2E test plan. In time, you’ll have a comfortable regression test for your UX. However, take the time to decide the testing approach and invest some time up front in developing common code and templates. Allow the person coding the test steps to focus on the test, not the idiosyncrasies of getting the browser to perform its tasks.

Chapter 11 Summary

Angular is a complete application development framework for creating component-based web applications. The testing environment used by Angular allows a developer or a team of developers to create a more robust application, with the goal of reducing bugs and providing a solid development and testing environment.

By using the various frameworks and tools, you can create a rich testing environment for yourself or a complete team. Take the time to set up and take advantage of these tools to reduce those late night, “uh oh, something broke” phone calls.

The following websites provide more information about the testing tools available to the Angular framework.

TSLint

The default linter installed by Angular CLI is TSLint, and it can be found at palantir.github.io/tslint. Keep in mind that while this is a powerful lint tool, it is deprecated in favor of ESLint.



ESLint is the lint tool that the developers of TypeScript have decided to use going forward. The author of TSLint has indicated support for the development effort of ESLint, and plans to deprecate TSLint. You can find out more about ESLint at eslint.org.

ESLint also supports plugins to allow further rule definitions. One such plugin is the ESLint plugin for Angular, available at github.com/EmmanuelDemey/eslint-plugin-angular.

If you install this plugin to your development environment, you will need to update your ESLint configuration file (`.eslintrc`) to include the following setting:

```
“plugins”: [ “angular”, “@typescript-eslint”]
```



[SonarQube](https://sonarqube.com) is a leading code-quality tool. It has over 100,000 users and can be used with 27 different languages. SonarLint is a lint checker you can integrate into your IDE, and can be found at sonarlint.org.

While some of the issues raised by SonarLint will be caught by other linting tools, SonarLint provides integration in the editor to allow you to find and fix the issues while you are developing your code.

Karma KARMA

Karma is a command line tool used to spawn a web server that can run your application test files. You can find information on the Karma application at karma-runner.github.io/0.12/index.html.

Jasmine

Jasmine is the open-source library that provides a structured set of methods to perform unit testing and E2E testing with the Angular framework. Documentation for the Jasmine framework can be found at jasmine.github.io/index.html.

You can see [Appendixes A](#) and [B](#) for some details on the Jasmine testing framework.

Protractor

Protractor is the end-to-end test framework used by Angular to run your E2E test suite. It allows your tests to be run in an actual browser, behaving as an end user would. You can find out more about Protractor at protractortest.org.

Protractor runs tests written in the Jasmine framework with Angular E2E testing.

Cucumber

Cucumber is a framework for behavior-driven development that allows you to test suites written in the Gherkin syntax. The **Given/When/Then** syntax of Gherkin is a popular language for business analysts to write feature files. You can learn about the framework at cucumber.io.

Although Cucumber and Gherkin are not the default test frameworks for Angular, they are popular solutions for behavior-driven development processes. [Appendix C](#) provides a list of the Gherkin keywords.

Summary

There is a large collection of testing tools and applications for the Angular development environment. We summarized some of the tools covered in the book in this chapter. Take the time to create your own development testing setup. It will greatly improve the quality of your Angular application and code.

Appendix A Jasmine Syntax

The Jasmine framework provides several functions to allow you to write test specifications. The basic structure of the test spec file follows:

```
describe(' What this module is testing ', () =>
{
  it('The service should be created', () => {
    const service: CustomerInfoService = TestBed.get(CustomerInfoService);
    expect(service).toBeTruthy();
  });
  it(' Should get a valid result ', () => {
    ...
    expect(ans).toBeDefined();
  });
  it(' Should test error conditions ', () => {
    ...
    expect(ans).toBeFalsy();
  });
}
```

Table A lists the functions that can be used as part of the Jasmine syntax to perform your tests.

Table A: Jasmine functions

Function	Description
describe()	Text description and list of tests (it functions) to run that make up the entire test suite.
fdescribe()	By adding the letter f to the front of a describe function, you can tell the runner to only run specific (focused) test suites.
xdescribe()	Marks a test suite as pending and not to be executed.
it()	Actual test specification. Contains a description string and the actual function to perform the test.
fit()	A focus it() definition. Only these tests should be run.

Function	Description
xit()	Marks a test as pending, not to be executed.
afterAll()	Defines code to be run after the entire set of tests has been run.
afterEach()	Defines code to be run after each test (it function) is executed.
beforeAll()	Code to be run before the entire set of tests is started.
beforeEach()	Code to run before each test (it function) is started.
expect()	Defines the expected results (using matchers) of the test.
expectAsync()	Asynchronous expectation match.
fail()	Indicates that a test specifically failed. Can optionally provide a message indicating why the test failed.
pending()	Ignore the expected comparison. Can optionally provide a string message indicating why the test is pending.
spyOn()	Installs a spy stub method onto an object.
spyOnAllFunctions()	Installs a spy on all functions within the specified object.
spyOnProperty()	Spies on a particular property and accesses type (get/set) of an object.

Appendix B Jasmine Matchers

The Jasmine testing framework provides many matchers that can be used for comparing results of the test with expected values. These matchers are described in Table B.

Table B: Jasmine matchers

Matcher	Description
not	Used to negate result of any matcher.
toBe()	Actual value and type must match (=== comparison).
toBeCloseTo()	Expected and precision, result must be close to.
toBeDefined()	Value must be defined (not undefined).
toBeFalse()	Value must be Boolean value false.
toBeFalsy()	Value must be falsy (false, null, empty string, undefined, NaN, 0).
toBeGreaterThan()	Value must be greater than specified numeric value.
toBeGreaterThanOrEqual()	Value must be greater than or equal to specified value.
toHaveClass()	DOM element must have specified class.
toBeInstanceOf()	Object must be an instance of expected class.
toBeLessThan()	Value must be less than the specified numeric value.
toBeLessThanOrEqual()	Value must be less than or equal to the expected value.
toMatch()	String value must match regular expression.
toBeNaN()	Value must be NaN (not a number).
toBeNull()	Value must be null.
toThrow()	An error must have been thrown. You can also provide a value that was expected to have been thrown.
toBeTrue()	Value must be Boolean value true.
toBeTruthy()	Value must be truthy (not a falsy value).
toBeUndefined()	Value must be undefined.

Matcher	Description
toContain()	If an array, the element must exist with the array. If a string, then the substring must exist within the string.
toEqual()	Actual value must be equal, using deep equality. (All properties of the objects are compared.)

You can expand the Jasmine-provided matchers using a third-party library [here](#).

This library adds many additional matchers, such as:

- **toBeArrayOfStrings()**
- **toHaveJsonString()**
- **toBeHtmlString()**

You can install the additional matchers using the following npm command line in your project folder:

```
npm install jasmine-expect --save-dev
```

Spy matchers

Jasmine also has functions called spies, which stub any function and track calls to the function. Table C lists the available spy matchers.

Table C: Jasmine spy matchers

Matcher	Description
toHaveBeenCalled()	The function has been called.
toHaveBeenCalledBefore()	The function had to be called prior to another spy function.
toHaveBeenCalledTimes()	The function has been called a specific number of times.
toHaveBeenCalledWith()	The function should have been called with specified parameters.

Appendix C Gherkin Syntax

The Gherkin syntax allows business analysts to write feature files in a consistent, readable language that developers can then use as a specification and test plan. Table D lists the basic keywords in the Gherkin syntax. While the table lists the keywords in English, the Gherkin syntax is available in over 70 languages.

Table D: Gherkin keywords

Keyword	Description
Feature	Free-form text to describe what the feature file test is about and give a high-level description of the feature. Begins after the Feature: syntax and ends when the first keyword is found in the file.
Example:	A list of steps (such as Given , When , and Then) to perform the test being done in this step. Should keep the list of steps small (3–5 steps). Too many steps will lose the focus of what the test is for. The keyword Scenario: can also be used, rather than Example: .
Given	This step is a precondition before the test starts. It might be to navigate to a webpage or provide a starting value for a variable, for example.
When	This keyword describes the action to be performed, such as I click a button, or I enter the value \$100 into the withdrawal amount.
Then	This step describes the expected outcome of the test. It should be something that can be observed, such as an error message is displayed, or the balance is reduced.
And, But	These keywords repeat the prior keyword, but allow the step definitions to be more readable. For example, the following steps are identical: Given I am on the main page screen Given I have successfully logged in Given I am on the main page screen And I have successfully logged in
Background	If you need to write the same Given (preconditions) for all steps in your file, you can attach those steps to a single Background keyword, and the step will be repeated for each Example or Scenario keyword in the suite.
Scenario Outline	Allows a single test to be run against a table of values. You define the steps using Given , When , and Then . Following the steps is a table structure of the values to repeat the test for.