**Chapter 14. Testing Angular applications**

*This chapter covers*

* Using the Jasmine framework for unit testing
* Identifying the main artifacts from the Angular testing library
* Testing services, components, and the router
* Running unit tests against web browsers with the Karma test runner
* End-to-end testing with the Protractor framework

To ensure that your software has no bugs, you need to test it. Even if your application has no bugs today, it may have them tomorrow, after you modify the existing code or introduce new code. Even if you don’t change the code in a particular module, it may stop working properly as a result of changes in another module or in the runtime environment. Your application code has to be retested regularly, and that process should be automated. You should prepare test scripts and start running them as early as possible in your development cycle.

This chapter covers two main types of testing for the frontend of web apps:

* *Unit testing***—**Asserts that a small unit of code accepts the expected input data and returns the expected result. Unit testing is about testing isolated pieces of code, especially public interfaces.
* *End-to-end testing***—**Asserts that the entire application works as end users expect and that all application parts properly interact with each other.

Unit tests are for testing the business logic of small, isolated units of code. They run reasonably fast, and you’ll be running unit tests a lot more often than end-to-end tests. End-to-end (e2e) testing simulates user actions (such as button clicks) and checks that the application behaves as expected. During end-to-end testing, you shouldn’t run unit-testing scripts.

##### NOTE

There are also integration tests that check that more that one app member can communicate. Whereas unit tests mock dependencies (for example, HTTP responses), integration tests use the real ones. To turn a unit test into an integration test, don’t use mocks.

We’ll start by covering the basics of unit testing with Jasmine, and then we’ll show you how the Angular testing library is used with Jasmine. After that, you’ll see how to use Protractor, the library for e2e tests. Toward the end of the chapter, we’ll show you how to write and run e2e scripts to test the product-search workflow of ngAuction.

### 14.1. Unit testing

The authors of this book work as consultants on large projects for various clients. Pretty often these projects were written without unit tests in place. We’re going to describe a typical situation that we’ve run into on multiple occasions.

A large app evolves over several years. Some of the developers who started writing the app are gone. A new developer joins the project and has to quickly learn the code and get up to speed.

A new business requirement comes in, and the new team member starts working on it. They implement this requirement in the existing function doSomething(), but the QA team opens another issue, reporting that the app is broken in a seemingly unrelated area. After additional research, it becomes obvious that the app is broken because of the code change made in doSomething(). The new developer doesn’t know about a certain business condition and can’t account for it.

This wouldn’t have happened if unit (or e2e) tests were written with the original version of doSomething() and run as a part of each build. Besides, the original unit test would serve as documentation for doSomething(). Although writing unit tests seems like an additional, time-consuming task, it may save you a lot more time in the long run.

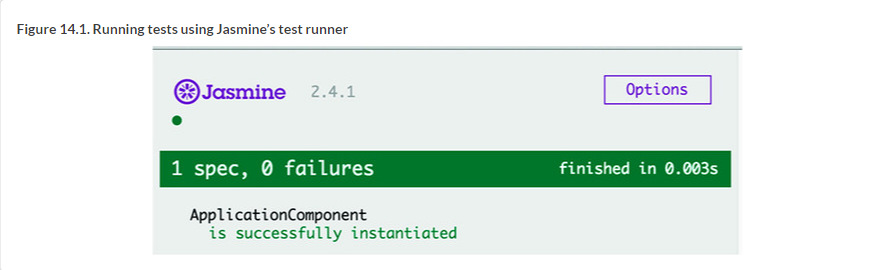
We like the definition given by Google engineer Elliotte Rusty Harold during one of his presentations—that a unit test should verify that a known, fixed input produces a known, fixed output. If you provide a fixed input for a function that internally uses other dependencies, those dependencies should be mocked out, so a single unit test script tests an isolated unit of code.

Several frameworks have been created specifically for writing unit tests, and Angular documentation recommends Jasmine for this purpose (see the Angular documentation at <http://mng.bz/0nv3>). We’ll start with a brief overview of Jasmine.

#### 14.1.1. Getting to know Jasmine

Jasmine (<https://jasmine.github.io/>) enables you to implement a *behavior-driven development* (BDD) process, which suggests that tests of any unit of software should be specified in terms of the desired behavior of the unit. With BDD, you use natural language constructs to describe what you think your code should be doing. You write unit test specifications (specs) in the form of short sentences, such as “StarsComponent emits the rating change event.”

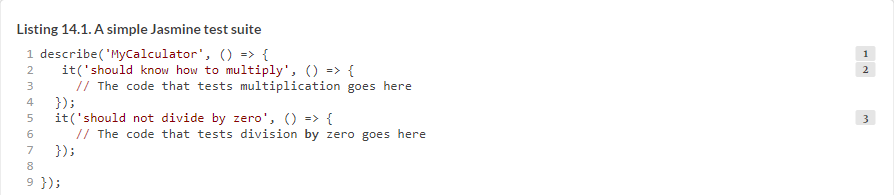
Because it’s so easy to understand the meaning of tests, they can serve as your program documentation. If other developers need to become familiar with your code, they can start by reading the code for the unit tests to understand your intentions. Using natural language to describe tests has another advantage: it’s easy to reason about the test results, as shown in [figure 14.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/19#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14fig01).



##### TIP

Even though Jasmine comes with its own browser-based test runner, you’ll be using a command-line-based test runner called Karma that can be easily integrated into the automated build process of your apps.

In BDD frameworks, a test is called a *spec*, and a combination of one or more specs is called a *suite*. A test suite is defined with the describe() function—this is where you describe what you’re testing. Each spec in a suite is programmed as an it() function, which defines the expected behavior of the code under test and how to test it. The following listing shows an example.



Testing frameworks have the notion of an *assertion*, which is a way of questioning whether an expression under test is true or false. If the assertion returns false, the framework throws an error. In Jasmine, assertions are specified using the expect() function, followed by *matchers*:toBe(), toEqual(), and so on. It’s as if you’re writing a sentence, “I expect 2 plus 2 to equal 4”:



Matchers implement a Boolean comparison between the actual and expected values. If the matcher returns true, the spec passes. If you expect a test result not to have a certain value, just add the keyword not before the matcher:



##### NOTE

You can find the complete list of matchers in the type definition file @types/jasmine/index.d.ts, located in the directory node\_modules. The Angular testing library adds more matchers, listed at <http://mng.bz/hx5u>.

In Angular, test suites have the same names as the files under test, adding the suffix .spec to the name. For example, the file application.spec.ts contains the test script for application.ts. [Figure 14.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/31#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14fig02) shows a minimalistic test suite that can be located in the app.component.spec.ts file; it makes an *assertion* that the variable app is an instance of AppComponent. An assertion is the expectation plus the matcher.



[Figure 14.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/37#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14fig02) shows a test suite containing a single spec. If you extract the texts from describe() and it() and put them together, you’ll get a sentence that clearly indicates what you’re testing here: “ApplicationComponent is successfully instantiated.” If other developers need to know what your spec tests, they can read the texts in describe() and it(). Each test should be self-descriptive so it can serve as program documentation.

##### TIP

Although the test shown in [figure 14.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/37#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14fig02) was generated by Angular CLI, it’s pretty useless because the chances that the AppComponent won’t be successfully instantiated are close to zero.

The code in [figure 14.2](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/37#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14fig02) instantiates AppComponent and expects the expression app instanceof AppComponent to evaluate to true. From the import statement, you can guess that this test script is located in the same directory as AppComponent.

##### NOTE

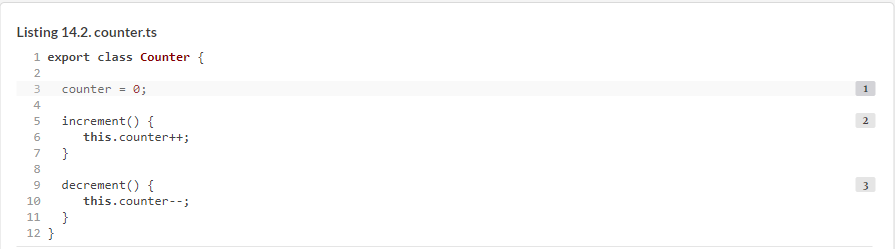
In Angular applications, you keep each test script in the same directory as the component (or service) under test, so if you need to reuse a component in another app, all related files are located together. If you use Angular CLI for generating a component or service, the boilerplate code for tests (the .spec.ts file) will be generated in the same directory.

If you want some code to be executed before each test (such as to prepare test dependencies), you can specify it in the *setup* functions beforeAll() and beforeEach(), which will run before the suite or each spec, respectively. If you want to execute some code right after the suite or each spec is finished, use the *teardown* functions afterAll() and afterEach().

Let’s see how to apply Jasmine API while unit-testing a TypeScript class.

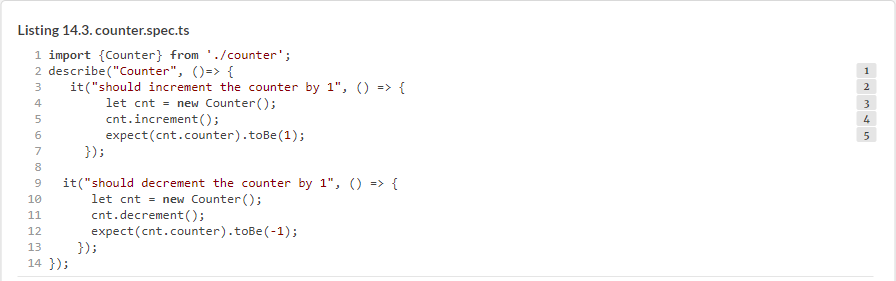
#### 14.1.2. Writing test scripts for a class

Imagine you have a Counter class with one counter property and two methods that allow incrementing or decrementing the value of this property.



What do you want to unit-test here? You want to make sure that the increment() method increments the value of counter by one, and that the decrement() method decrements this value by one. Applying Jasmine terminology, you want to write a test suite with two specs.

Remember that a spec should test an isolated piece of functionality, so each spec should create an instance of the Counter class and invoke *only one* of its methods. The first version of the counter.spec.ts file is shown in the following listing.



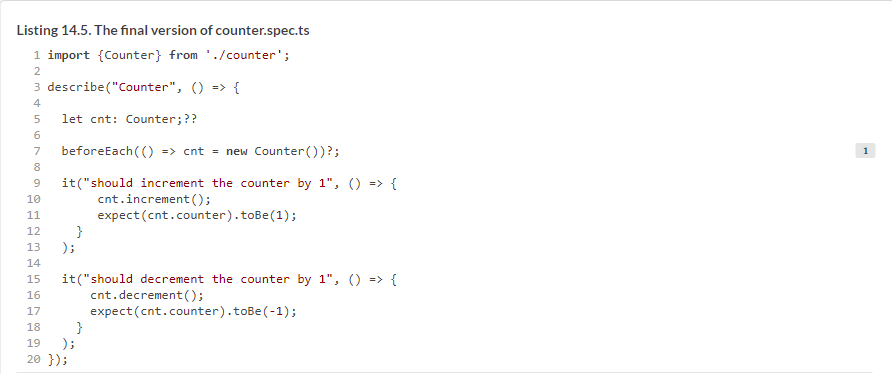
Each of your specs has similar functionality. The setup phase creates a fresh instance of the Counter class, then it invokes the method to be tested, and finally it declares the expectation with the expect() method. In one spec, you expect the counter to be 1, and in another -1.

This suite of tests will work, but you have some code duplication here: each of the specs repeats the instantiation of Counter. In the refactored version of your test script, you’ll remove the Counter instantiation from the specs and do it before the specs. Take a look at the new test version in the following listing. Is it correct?



This test is not correct. Your test suite will create an instance of Counter, and the first spec will increase the counter value to 1 as expected. But when the second spec decrements the counter, its value becomes 0, though the matcher expects it to be -1.

The final version of your test script, shown in the nest listing, fixes this mistake by creating the instance of Counter inside Jasmine’s beforeEach() function.



Now this script properly instructs Jasmine to create a new instance of Counter before running each spec of your suite. Let’s see how to run it.

### 14.2. Running Jasmine scripts with Karma

For projects that don’t use Angular CLI, you need to do lots of manual configurations to run Jasmine tests. Without Angular CLI, you start with installing Jasmine and its type definition files as follows:



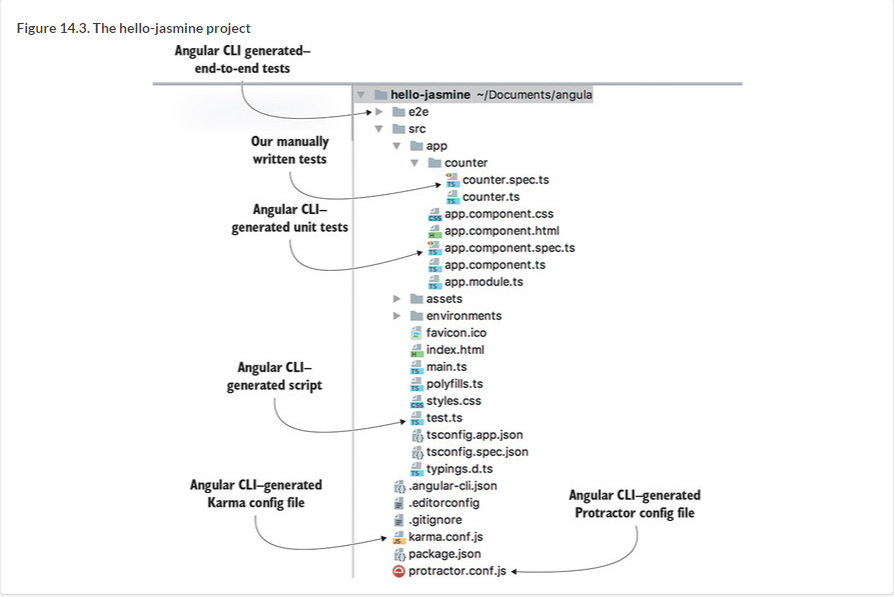
Then you need to create a test.html file that includes script tags to load Jasmine and your specs (the TypeScript code needs to be precompiled into JavaScript). Finally, you need to manually load test.html in each browser you care about and watch whether your tests fail or pass.

But running unit tests from the command line is a better option, because that way you can integrate tests into the project build process. This is one of the main reasons for using a command-line test runner called Karma (see [https://karma-runner.github.io](https://karma-runner.github.io/)). Along with that benefit, Karma has multiple useful plugins and can be used with many JavaScript testing libraries for testing against all major browsers.

Karma is used for testing JavaScript code written with or without frameworks. Karma can run tests to check whether your application works properly in multiple browsers (Chrome, Firefox, Internet Explorer, and so on). In non-Angular CLI projects, you can install Karma and the plugins for Jasmine, Chrome, and Firefox, as shown in the following listing.



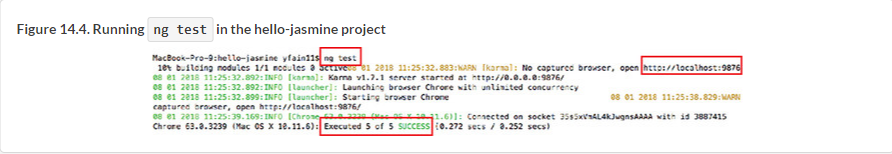
Then you need to prepare a configuration file, karma.conf.js, for your project—but you’re spoiled by Angular CLI, which installs and configures everything you need for testing Angular apps, including Jasmine and Karma. We’ve generated a new project with Angular CLI and added the code described in the previous section to test the Counter class there. You’ll find this project in the hello-jasmine directory. [Figure 14.3](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/63#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14fig03) shows the structure of this project, marking all test-related files and directories.



At the very top, you see the e2e directory, and at the bottom, the protractor.conf.js file, which were generated for end-to-end testing, described in [section 14.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/67#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14lev1sec4).

The counter.spec.ts file is the manually written test script described in the previous section. The app.component.spec.ts file was generated by Angular CLI for testing the AppComponent, and you’ll see its content in [section 14.3.1](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/67#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14lev2sec5).

The generated file test.ts is the main testing script that loads all test scripts. The karma.conf.js file is used by the Karma runner as soon as you run the ng test command, which compiles and runs unit tests. After the tests are compiled, ng test uses the compiled script test.js to load the Angular testing library and all the .spec.ts files, and start the Karma runner. [Figure 14.4](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/67#!/book/angular-development-with-typescript-second-edition/chapter-14/ch14fig04). shows the output of the ng test command that in the hello-jasmine project.



To run the tests, Karma starts the Chrome browser (the only one configured by Angular CLI) and runs five tests that end successfully. Why five? You wrote only two tests in the counter.spec.ts file, right? Angular CLI also generates the app.component .spec.ts file, which includes the test suite with three it() functions defined. Karma executes all files that have an extension .spec.ts.

##### NOTE

Angular CLI projects include the karma-jasmine-html-reporter package, and if you want to see the test results in the browser, open the URL http://localhost:9876.

You don’t want to run tests from app.component.spec.ts at this point, so let’s turn them off. If you want the test runner to skip some tests, rename their spec function from it() to xit(). Here, *x* is for *exclude*. If you want to skip the entire test suite, rename describe() to xdescribe().

If you exclude the test suite in app.component.spec.ts, the tests will be automatically rerun, reporting that two tests ran successfully (those that you wrote for Counter), and three specs were skipped (those that were generated by Angular CLI):

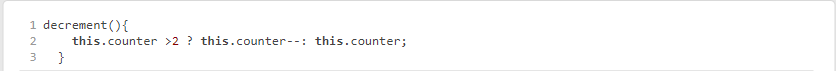


As the number of specs grows, you may want to execute just some of them to see the results faster. Renaming a spec function from it() to fit() (*f* is for *force*) will execute only these tests while skipping the rest.

##### YOU KNOW HOW TO TEST, BUT WHY IS STILL NOT CLEAR

Let’s say you know how to test the methods of your Counter class, but you still may have a million-dollar question: Why test such simple functions like increment() and decrement()? Isn’t it obvious that they’ll always work fine? In the real world, things change, and what used to be simple becomes not so simple anymore.

Say the business logic for the decrement() function changes, and the new requirement is not to allow counter to be less than 2. The developer changes the decrement() code to look like this the following.



Suddenly, you have two possible *execution paths*:

* The current counter value is greater than 2.
* The current counter value is equal to 2.

If you had the unit test for decrement(), the next time you run ng test it would fail, as follows:

* *1***The text describes the spec that failed.**
* *2***The assertion failed because the code under test didn’t decrement the counter that was equal to zero.**

The fact that your unit test failed is a good thing, because it tells you that something changed in the application logic—in decrement(). Now the developer should see what changed and add another spec to the test suite so you have two it() blocks testing both execution paths of decrement() to ensure that it always works properly.

In the real world, business requirements change pretty often, and if developers implement them without providing unit tests for the new functionality, your app can become unreliable and will keep you (or production support engineers) awake at night.

##### TIP

The output of the failed test may not be easy to read because it can include multiple lines of error stack trace. Consider using the continuous testing tool called Wallaby (see <https://wallabyjs.com/docs>), which shows you a short error message in your IDE right next to the code of the spec that failed.

##### NOTE

In [chapter 12](https://livebook.manning.com/book/angular-development-with-typescript-second-edition/chapter-14/91#!/book/angular-development-with-typescript-second-edition/chapter-12/ch12), section 12.3.6, we explained how to automate the build process by running a sequence of npm scripts. If you add ng test to your build command, the build will be aborted if any of the unit tests fail. For example, the build script can look like this: "build": "ng test && ng build".

It’s great that Angular CLI generates a Karma config file that works, but sometimes you may want to modify it based on your project needs.

#### 14.2.1. Karma configuration file

When Angular CLI generates a new project, it includes karma.conf.js preconfigured to run Jasmine unit tests in the Chrome browser. You can read about all available configuration options at <http://mng.bz/82cQ>, but we’ll just highlight some of them that you may want to modify in your projects. The generated karma.conf.js file is shown in the following listing.

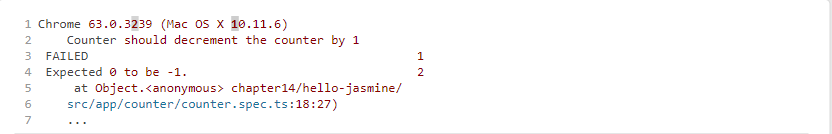


##### NOTE

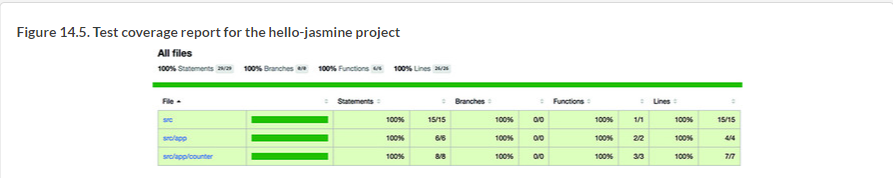
If you want Karma to print a message about each completed spec on the console, add karma-mocha-reporter as devDependency in package.json, add the line require('karma-mocha-reporter') to karma.conf.js, and replace the progress reporter with mocha. If you run tests in continuous integration (CI) servers, use the karma-junit-reporter that can write test results into a file in JUnit XML format.

This configuration file uses only the Chrome plugin, but in real-world apps, you want to run tests in several browsers. The next section shows you how to add Firefox to the list of browsers to be used in tests.

Karma can report how well your code is covered with tests using the Istanbul reporter, and you can run the following command to generate the coverage report:







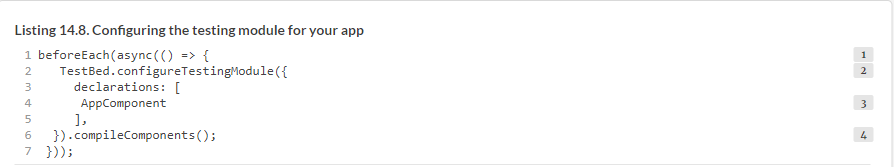
#### 14.2.2. Testing in multiple browsers



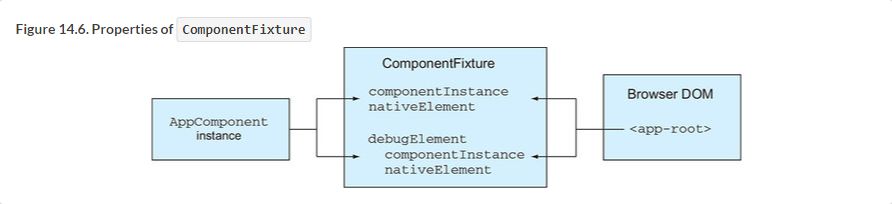




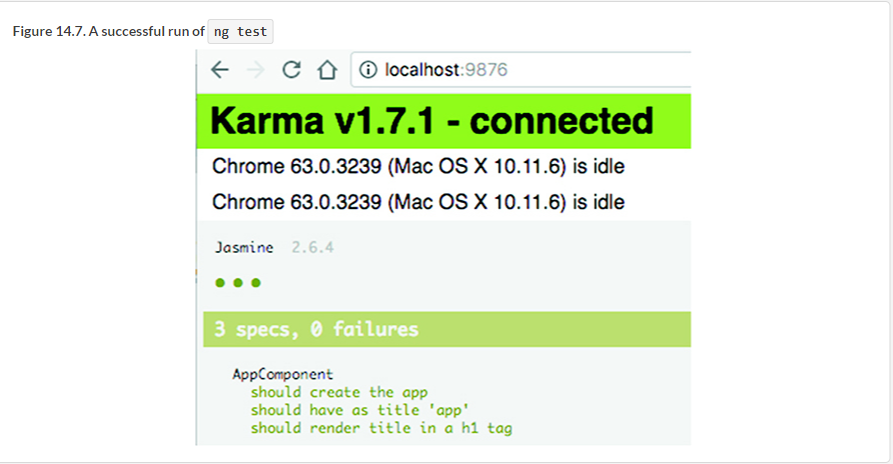
### 14.3. Using the Angular testing library

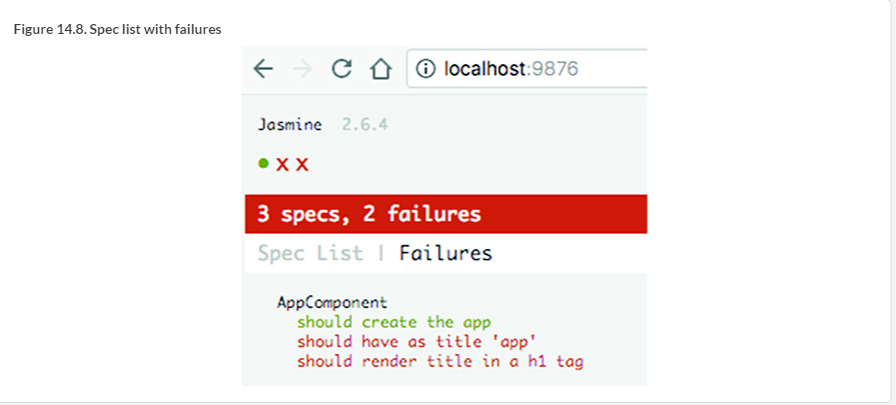


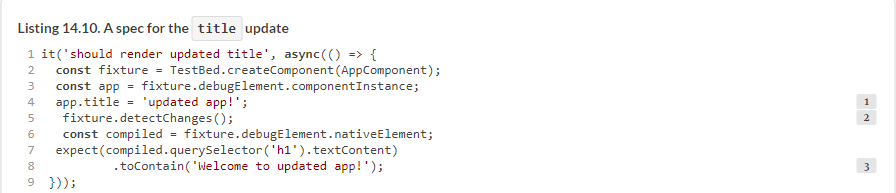
#### 14.3.1. Testing components



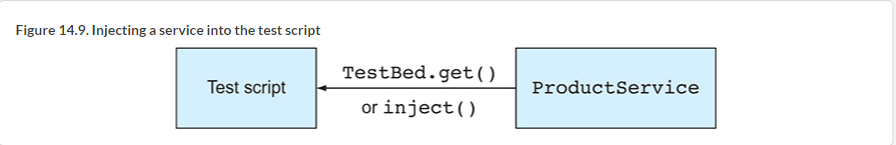








#### 14.3.2. Testing services



Component-level injectors can be used as follows:

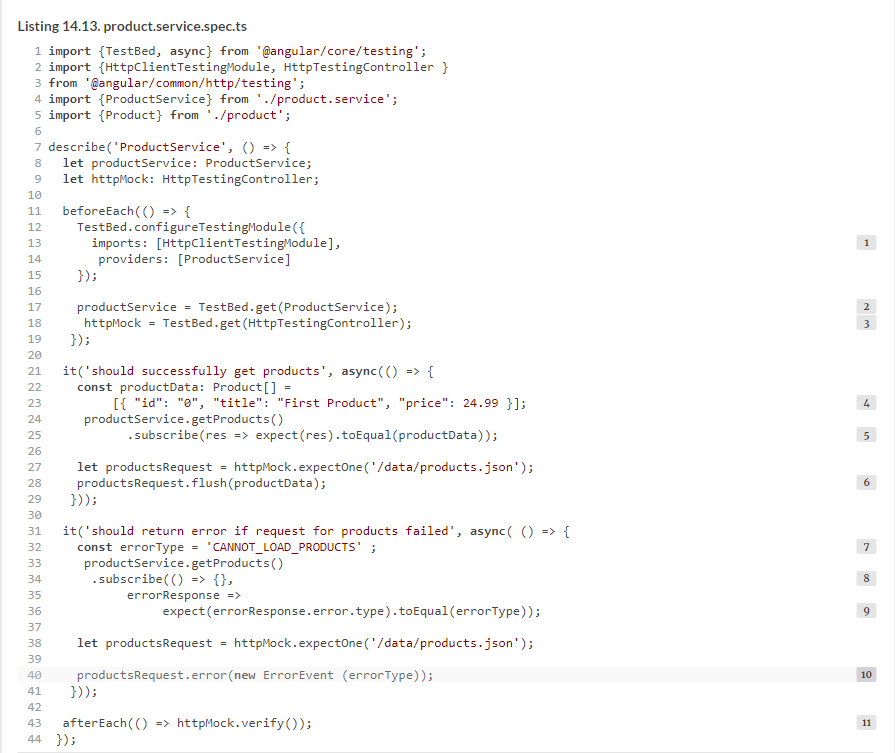










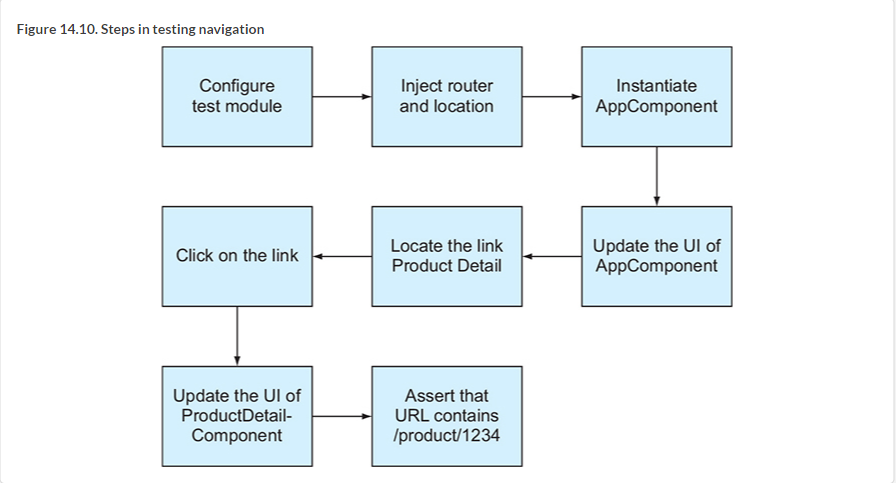


#### 14.3.3. Testing components that use routing



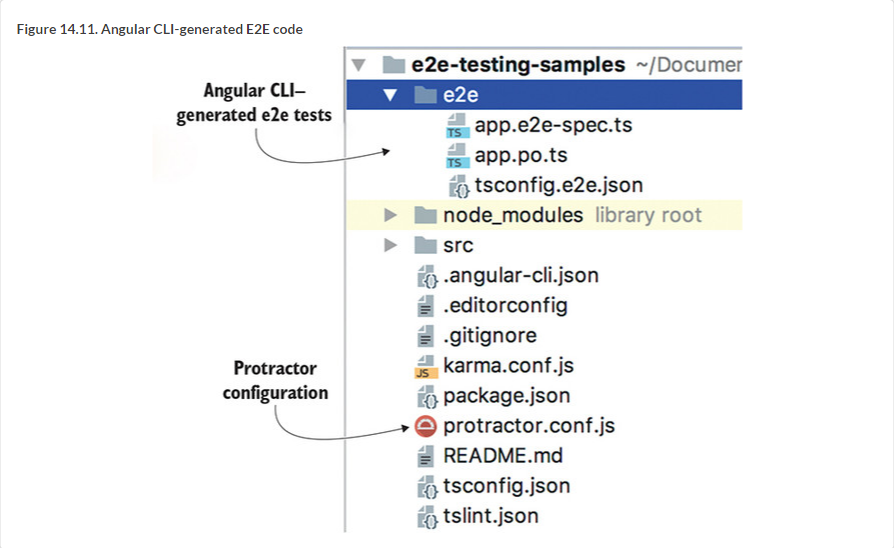


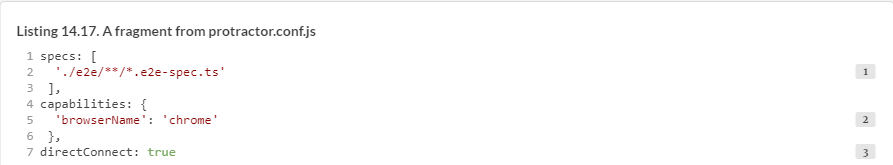


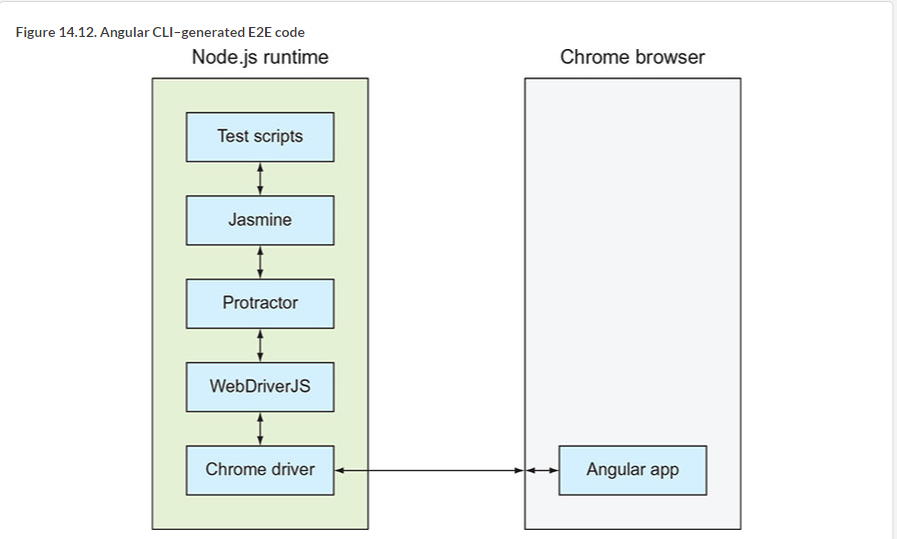


### 14.4. End-to-end testing with Protractor

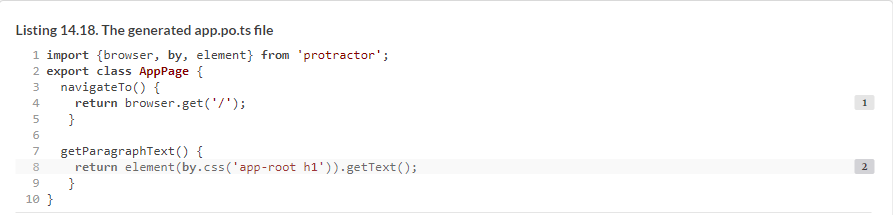
#### 14.4.1. Protractor basics

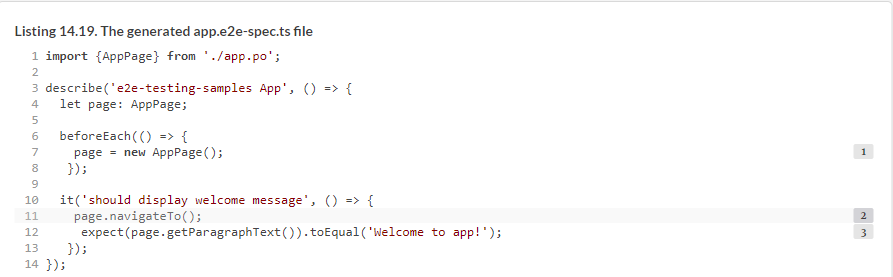




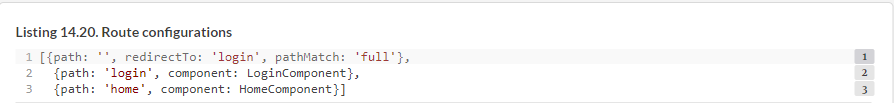


#### 14.4.2. Angular CLI–generated tests

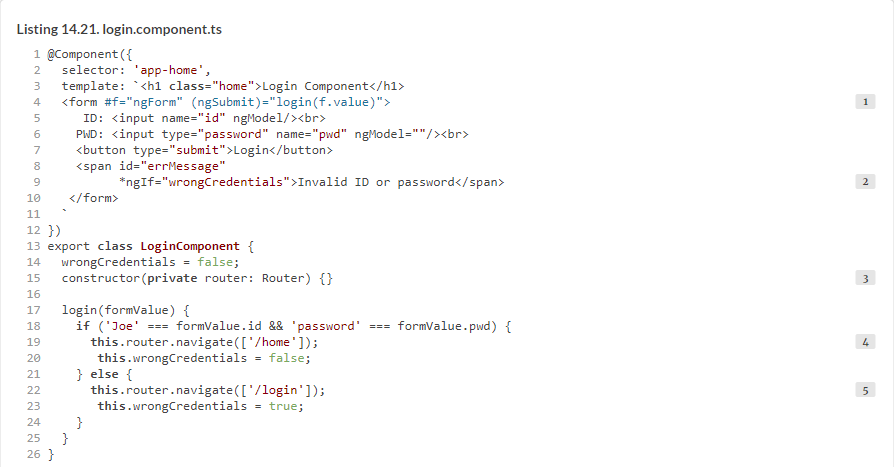




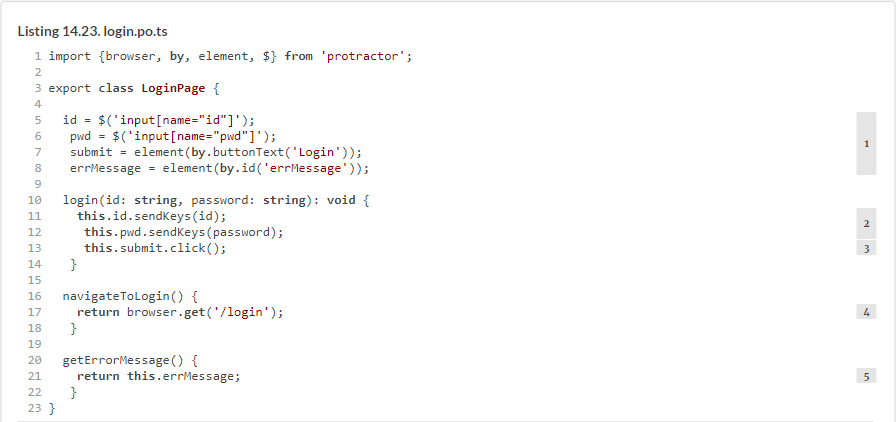
#### 14.4.3. Testing a login page



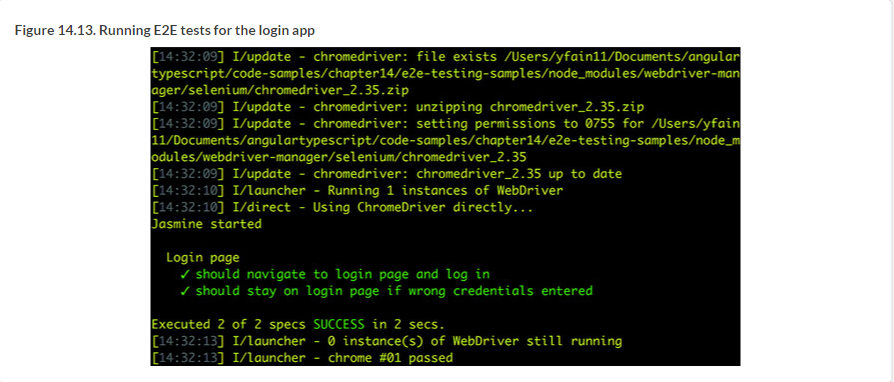


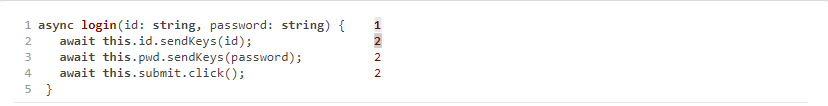










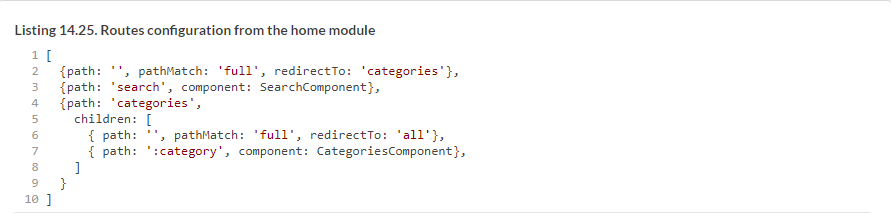


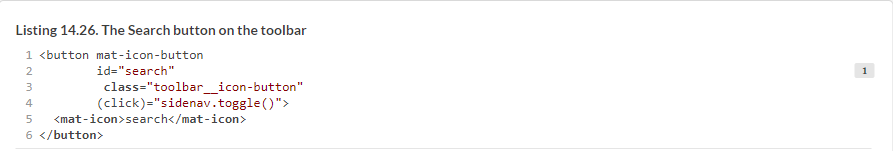


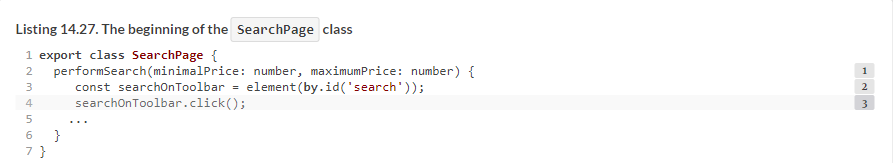
### 14.5. Hands-on: Adding an E2E test to ngAuction

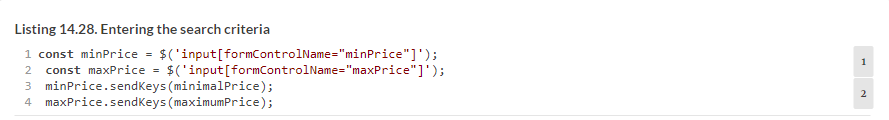


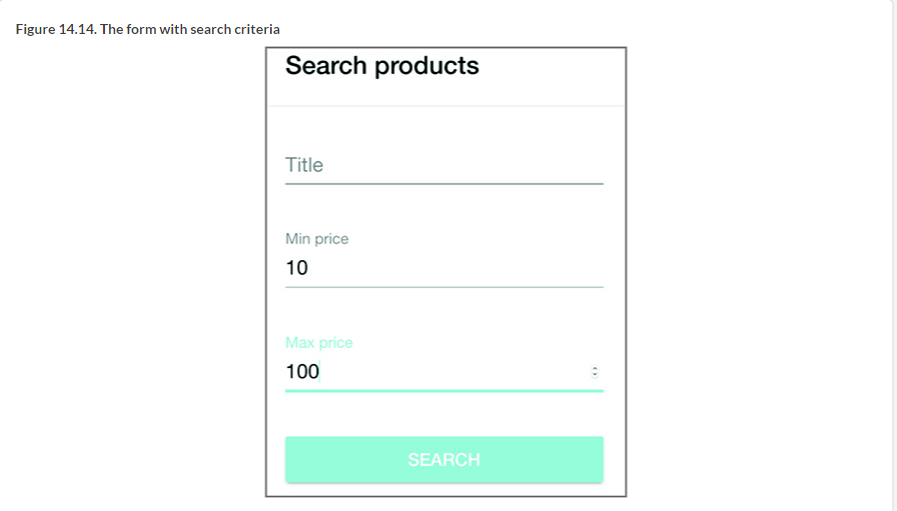
#### 14.5.1. E2E testing of the product-search workflow

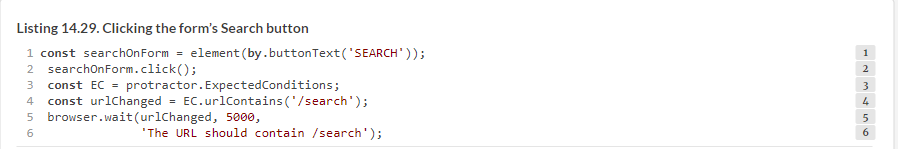


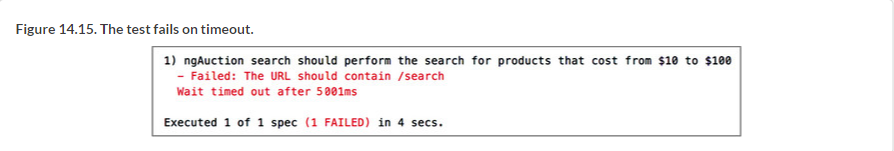


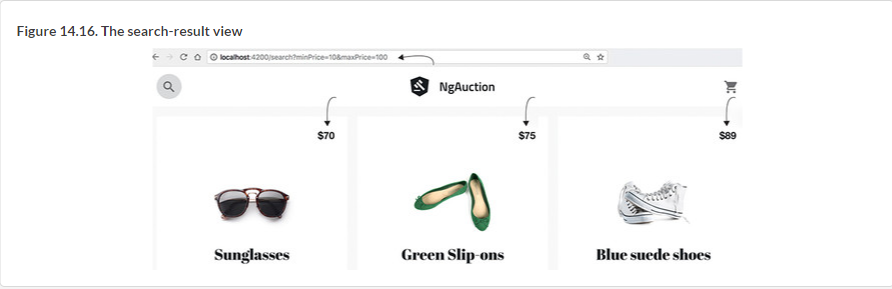


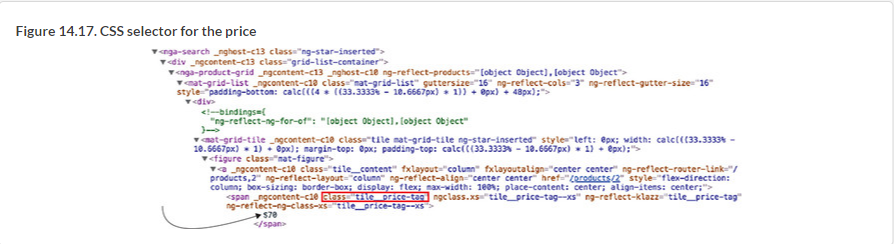


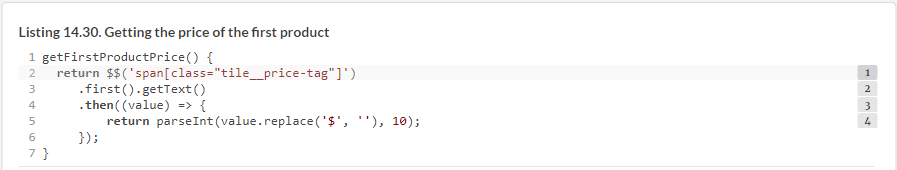




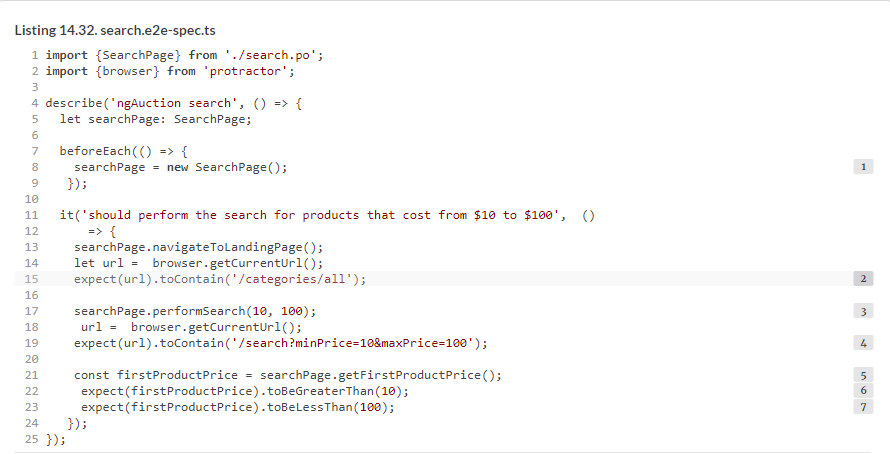


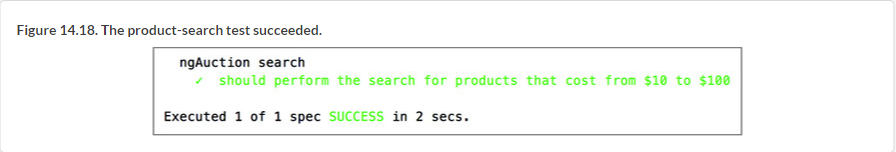












### Summary