**KARMA TESTING**

**Code coverage:** It is the amount of code that is being tested.

**Ng test –code-coverage**

It will create a new folder coverage and in that folder, there will be an index.html file. And if we open that in the browser we will get a summary of code coverage.

**1.Types of Tests:**

* 1. **Unit Tests:** In unit test, we test a component in isolation, without external resources (e.g. file system, database, API endpoints).

In angular terms, we test a component in isolation without it’s template and other external resources.

If a component is taking to a service, we give a fake instance of the service ‘FAKESERVICE’, assume that the service is working properly and focus only on the functionality of the component.

If a component is using a router, we give it a fake router.

**Uses:**

1. Easy to write.
2. Superfast.
3. Does not give enough confidence.
   1. **Integration Tests:** In integration test, we test a component with external resources.

In angular terms, we test a component along with its template. So, we need to run our test in an angular environment.

Even here, if a component is taking to a service, we give a fake instance of the service ‘FAKESERVICE’, assume that the service is working properly and focus on the functionality of the component along with the template.

If a component is using a router, we give it a fake router.

* 1. **End-to-end Tests:** In end-to-end test, we test the whole application together.

**2. Fundamentals of Unit Tests:**

Use the command “ng test” to launch KARMA.

We use describe () in unit testing to describe a suite, which is a group of related tests.

We use it () in unit testing to define a spec, which is an individual test.

To disable any test, just put an x before the it. so it() becomes xit() and the test is disabled.

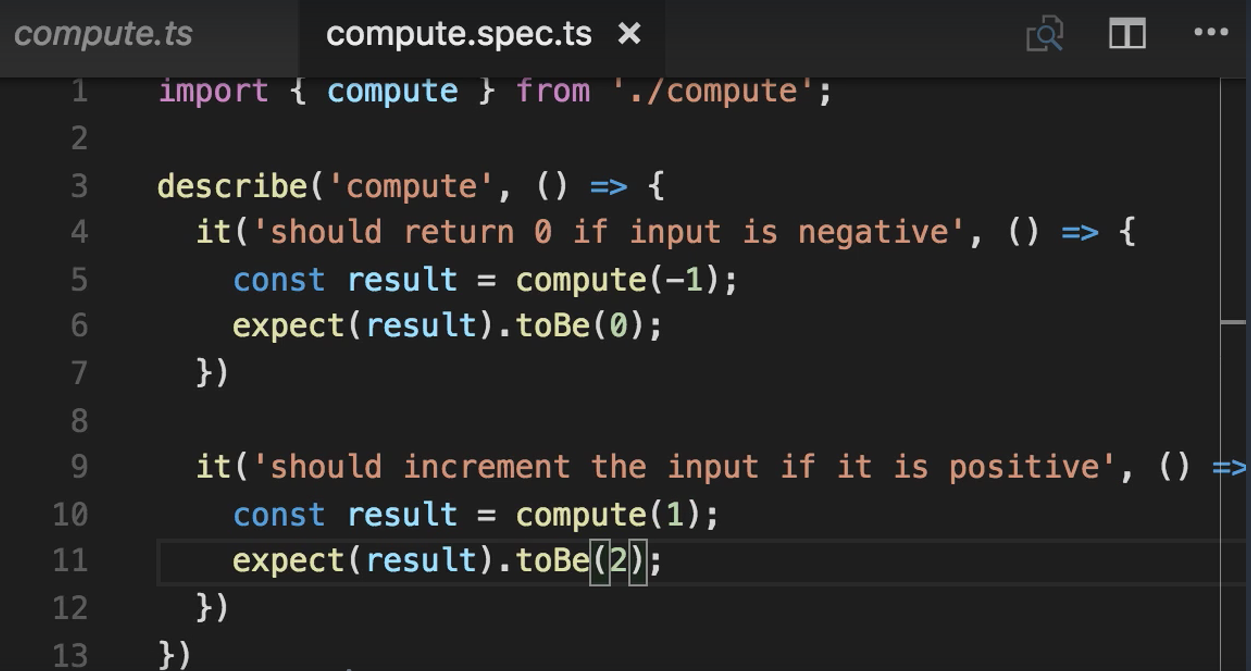
Syntax:

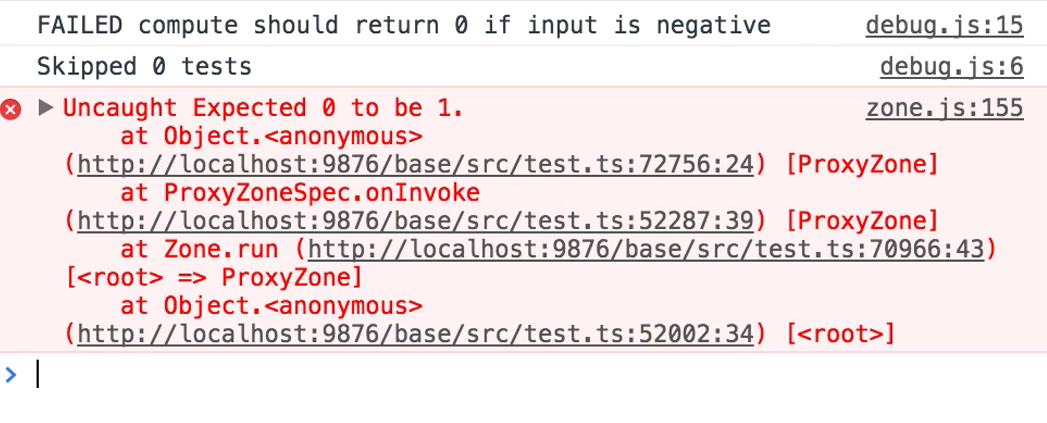
|  |
| --- |
| **describe (‘name of the system under test’, () => {**  **it (‘test name’, () => {**  **}**  **}** |

**2.1 UT on Simple methods:**

|  |
| --- |
| **Method: Written in compute.ts file**  export function compute(number) {  if(number<0) {  return 0;  }  return number++;  } |

**Test: Written in compute.spec.ts file**





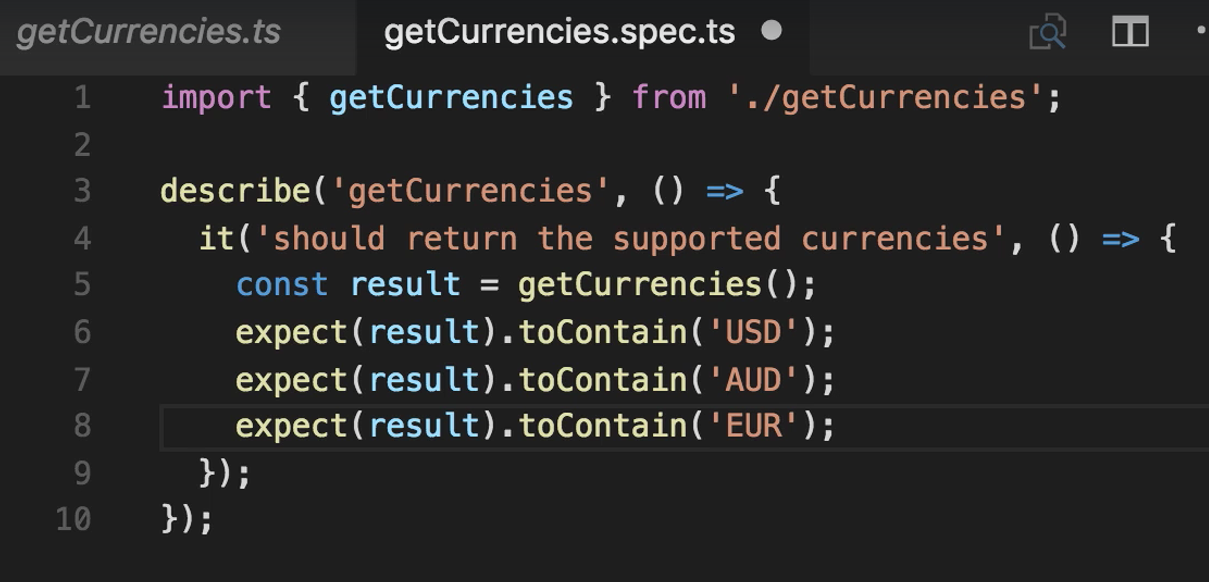
**2.2 UT on Strings:**

The return type of the methods (here greet) to be tested should be a string.



**2.3 UT on Arrays:**

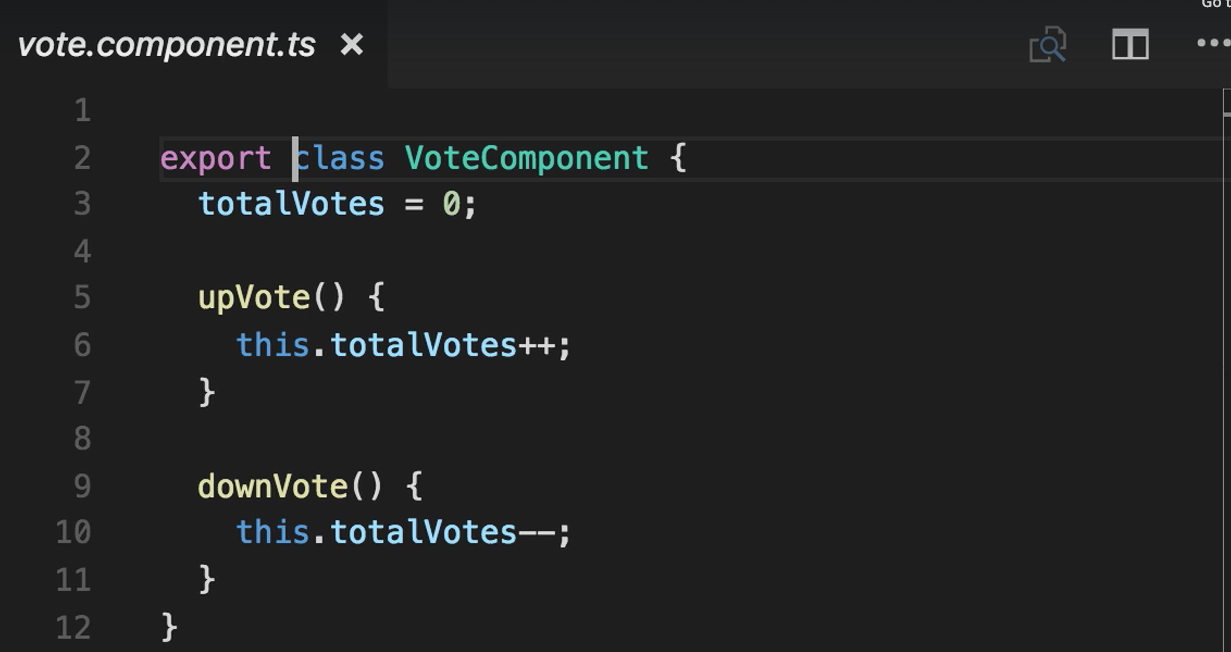
The return type of the methods (here getCurrencies) to be tested should be a string.

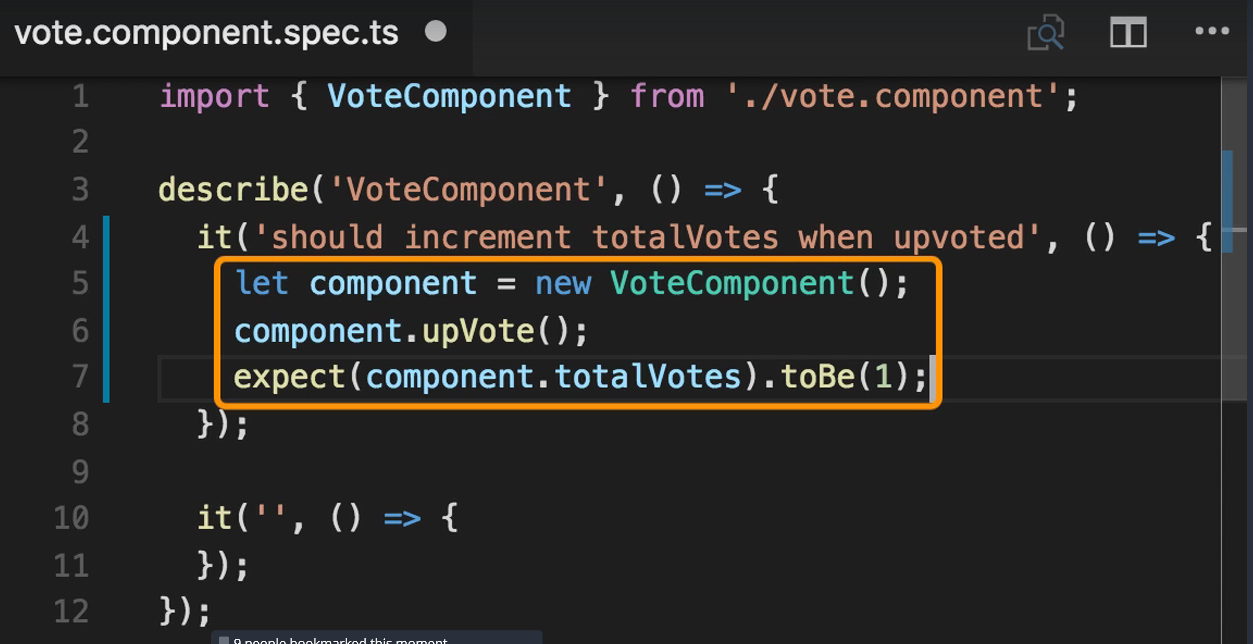


**2.4 UT on Components:**

A component can have many methods written in it. And testing can be done on the individual methods using instance of that component.

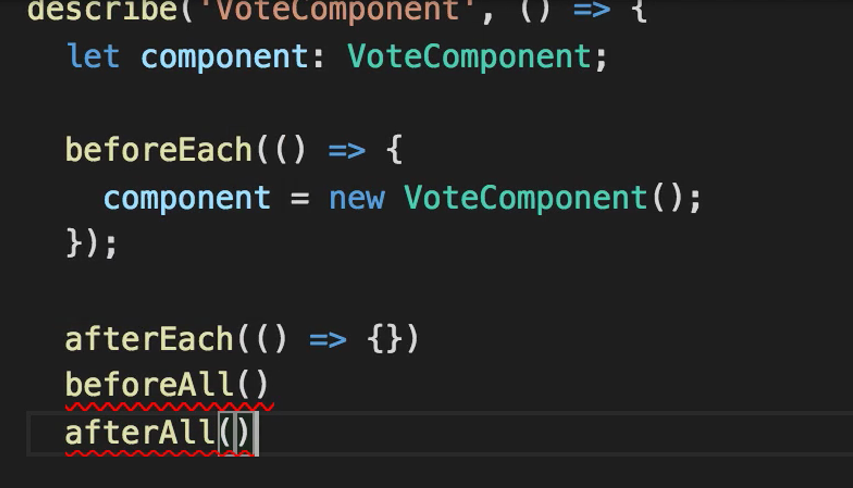
For example, below is a component





What if, we had to write multiple tests in a component on different methods. Then we must create instances in each test case.

Note: When writing tests, each test should be isolated and function as if it is the only test for that component. So, start each test with a clean state. That can be done by using the following functions.



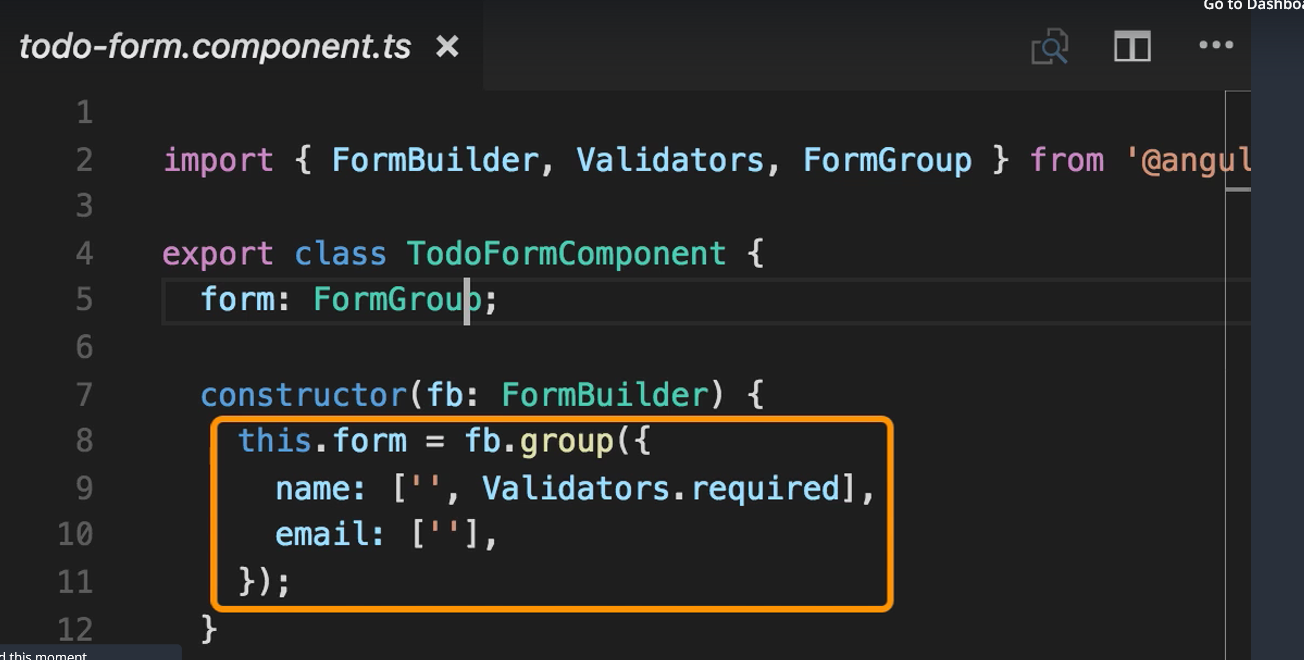
**BeforeEach () is executed before each test spec (test case). This is called Setup.**

**AfterEach () is executed after each case. Here we write the cleanup code. This**

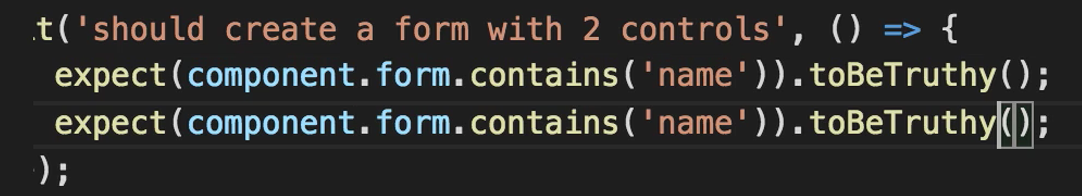
**BeforeAll () is executed once before all test spec (test case).**

**AfterAll () is executed once after all test spec (test case).**

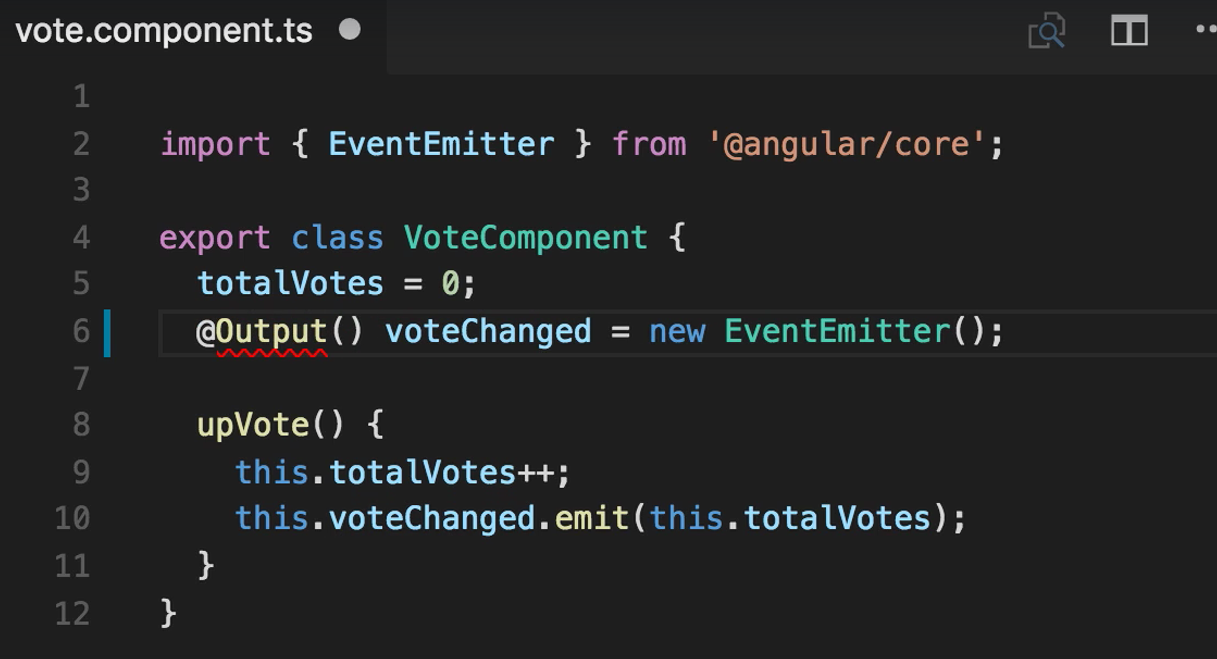
**2.5 UT on FORMS (Reactive Type):**



The test should check if the two fields are present and if the validators is present.



**2.6 UT on Event Emitters:**



**Event emitters are Observables. So, we must subscribe to that event to test it.**



**2.7 UT in Services:**

As in Unit Testing we should not talk to external resources, we create a Fake Service and perform operations.

This is how a service has been called in a component.





Note: In UT, we do not work with the instance of ServiceComponent ‘service’ but we can use the todos parameter and do unit testing.



We are not (should not) using the service here. That is why we are passing null to create a Fake service. In order not to call the server, we can use the **spyOn ()** method.

Testing ngOnInIt()



The callFake method is a replica of the method in the service. So, it was ‘**getTodos’** method in the service and it is the callFake method in the test.

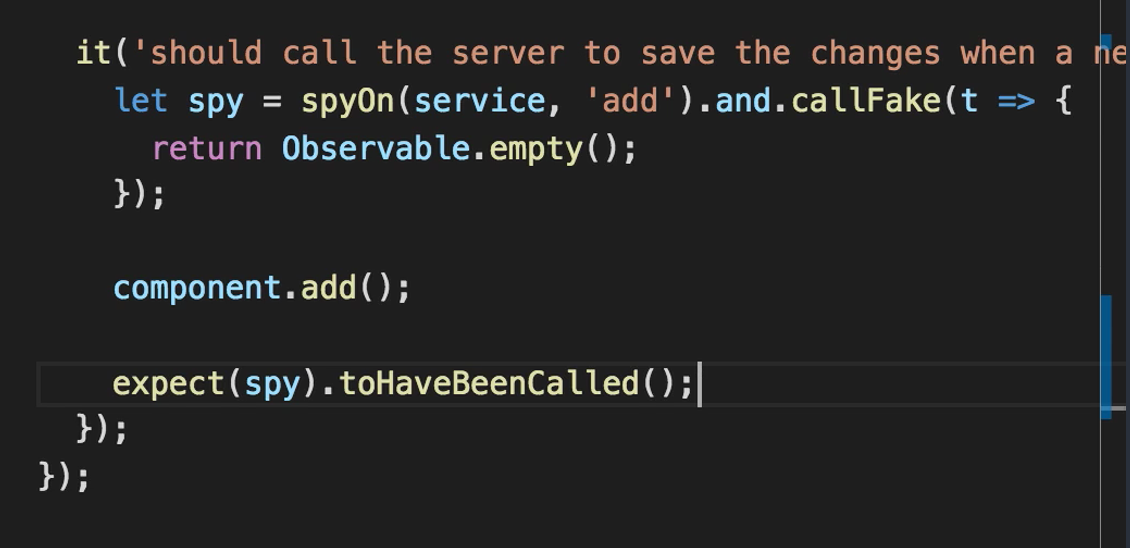
Testing the add method

In the add method we should check,

1. If the component is going to call the server to make changes. **(this. service. add)**
2. If it is added, is it pushing to the server.
3. If not, is the error message displayed.

Test 1:

Observable.empty() method is used.



Test 2:

Observable.from() method is used.



Instead of using the callFake method, and passing the value and returning the observable, we can use the returnValue function and pass the return statement as argument to this method. It should work the same. It would look like this

**Let spy = spyOn (service, ‘add’).and.returnValue(Observable.from([todo]));**

Test 3:

Observable.throw() method is used

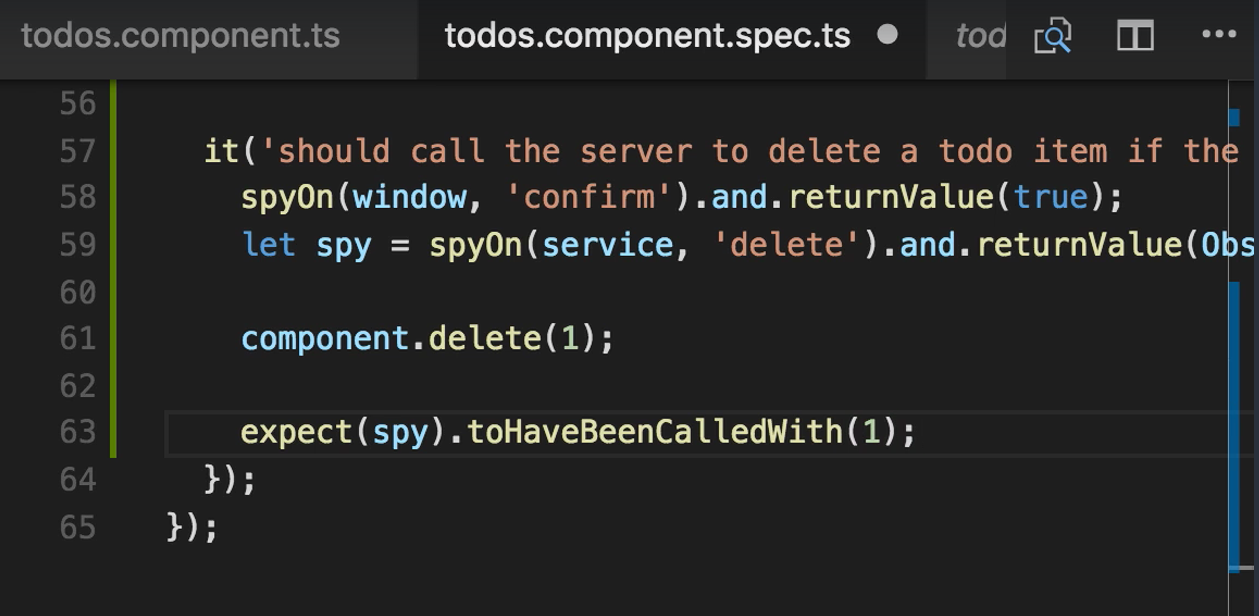


Test for deleting:

There are two tests to be performed.

1. Did the user confirm and did the request go to the server?
2. Did the user cancel and did it go to the service?

Test 1:



Test 2:

