Client-Side Unit Testing

# Types of Automated Testing

1. Unit Testing:
   1. Against a single unit of code
   2. Example: testing of one component
2. End to End Testing
   1. A live running application
   2. Testing the entire application
3. Integration Testing
   1. Two units of the application
   2. Testing the interaction between component and service
   3. Angular Integration test between template and component typescripts

# Mocking

Makes sure you are testing one unit of code at a time.

For any dependencies that you don’t want to test

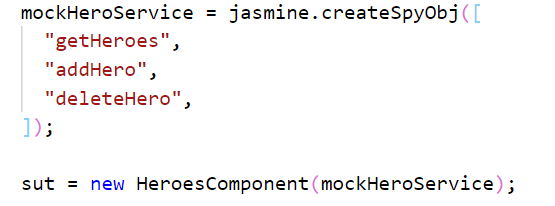
It’s a class that mocks an existing class or a service but we control what it does and what it returns

## Types of Mocks

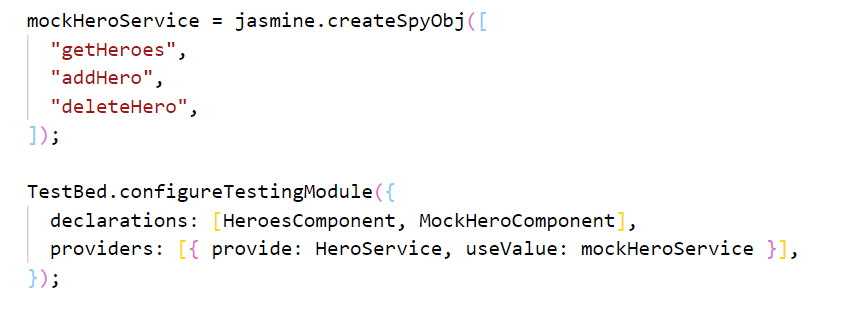
* Dummies:
  + Objects that fill a place.
  + No actual need for it.
  + Like passing a parameter to a method and we don’t care about the parameter.
* Stubs:
  + Object that has controllable behavior
  + We can decide what a method call will return from a Stub
* Spies:
  + Keeps track of which of its methods were called and how many times they were called
* True Mock
  + Verify that they were used in a specific way
  + Verify a method was called and only was called once and had a specific parameters

## Methods to mock

* We can create a mock object and pass it to constructor



* Add mock object to providers in the test module



# Types of unit tests in Angular

* Isolated:
  + Either to test the class of component, service, pipe
* Integration
  + We test one component, but we put it in a module to test it
  + So we can test the component with its template
  + Types:
    - Shallow: test only the component itself
    - Deep: test the component and child components and how they interact together

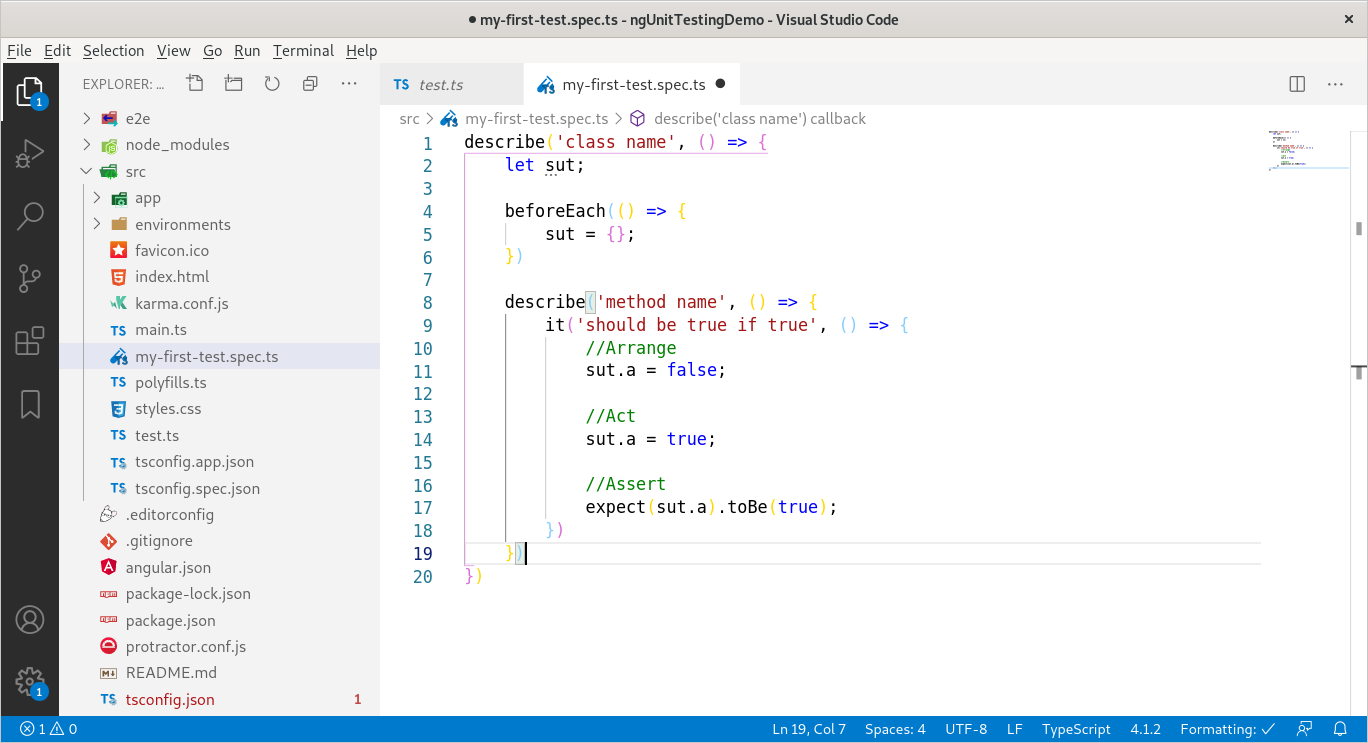
# Tools

Karma: Test Runner for Angular. Executes the tests in the browser.

Jasmine/Jest: To create the tests using Mocks or Expectations.

# Writing Tests

In order for Karma to know the file is unit tests it must be .Spec.ts



When writing tests make sure to follow that structure, it produces output like

class name method name should be true if true

For example, User Search Component Get Users Method Should Return No Results If No Filter

We can nest as many describe as we want

## Damp vs Dry

Damp means repeat yourself if you need to

## State Test

We test that the component state has changed

For example: element has been added or removed from array

## Interaction test

We test that an interaction has happened between the class we are testing and another class (service)

For example:

When method is called, the add method is called on service with correct parameter

## Shallow integration tests

Test the component only and none of its children or directives. We create fake or mock child components for testing.

We need to create a special module just for testing

* Test Bed: allows to test the component with the template

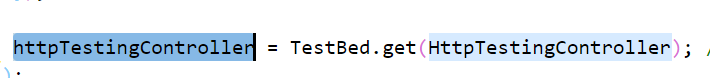
## Deep integration tests

Test the component interaction with its children. We check that data is passed to child component and set correctly inside it (input).

It also applies to testing services interaction with one another

## Mock of HttpClient

1. Need to add HttpClientTestingModule into the testing module imports
2. Need to add HttpTestingController: it allows up to mock how methods of http client behave

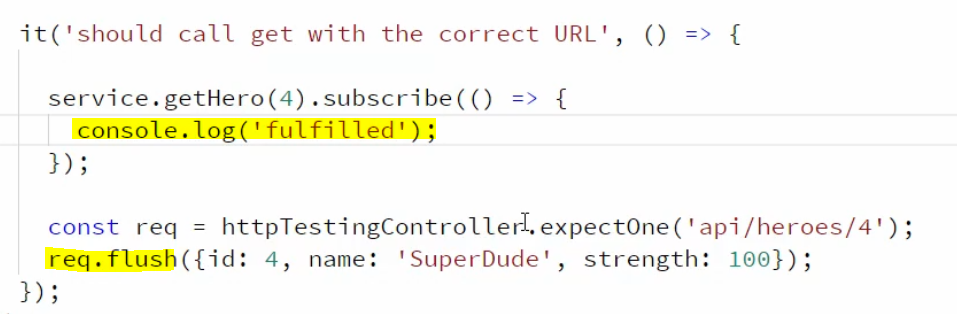


gets an instance for that type for dependency injection.

### Flush

To return something back from http request

### Order of execution

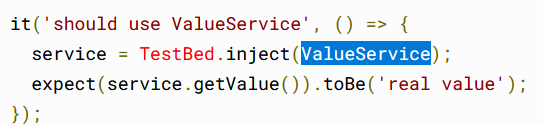


1. Service.getHero is called
2. ExpectOne is called
3. Flush is called
4. Subscribe callback is called

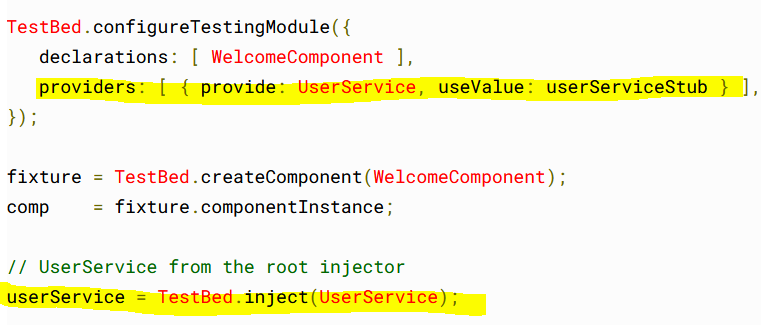
## Dependency injection for tests

To get a handle for dependencies that we want to use inside of our test we have two ways:

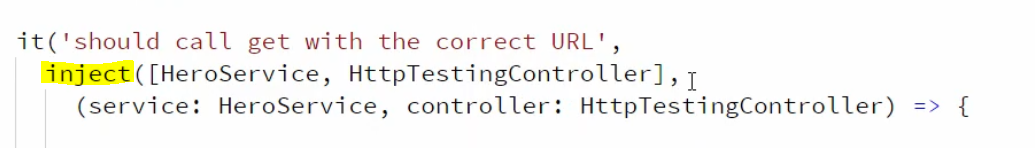
1. Use TestBed.inject(type): this will return an instance from the type.
   1. TestBed.get has depreciated from Angular 9
   2. However both return a real object of the service after resolving all dependencies
   3. It returns an object from root injector



The following example will return the stub because it was added in providers, so inject will return whatever we setup in the testing module



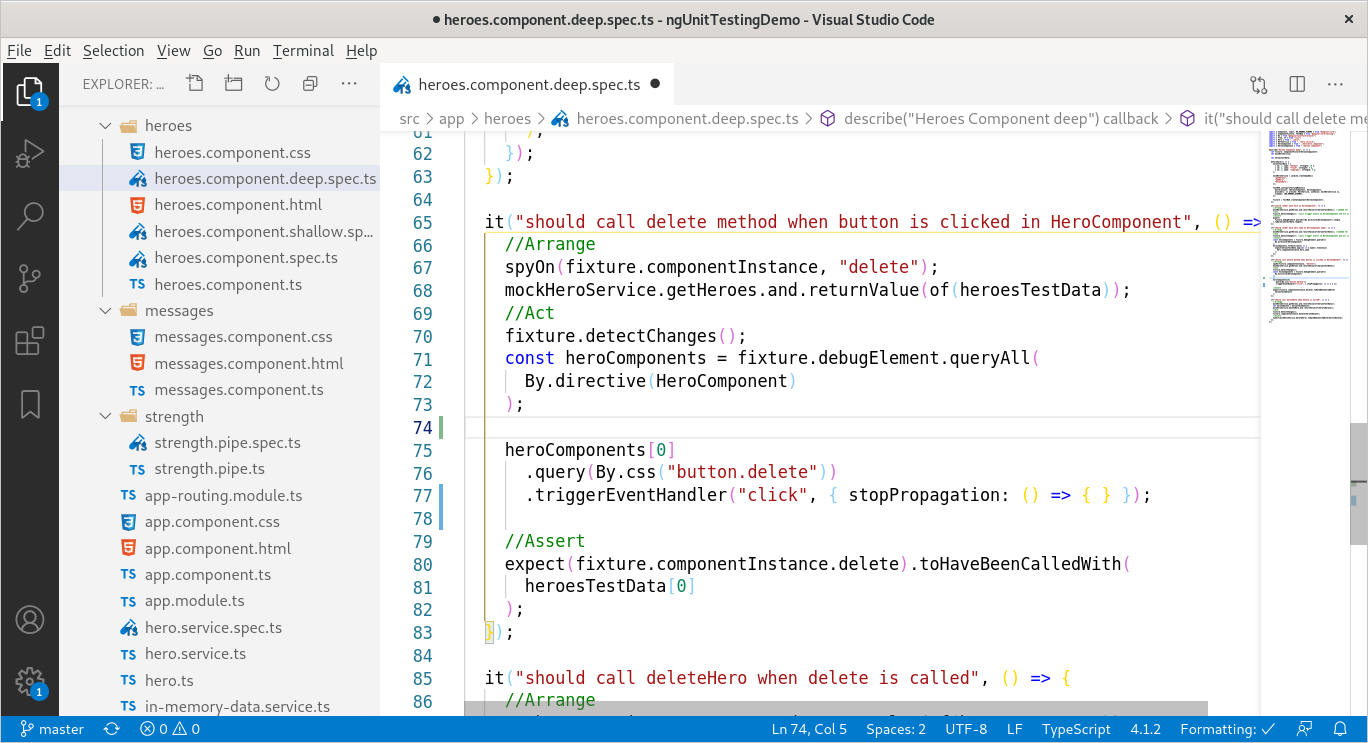
1. Use Inject method instead of callback:



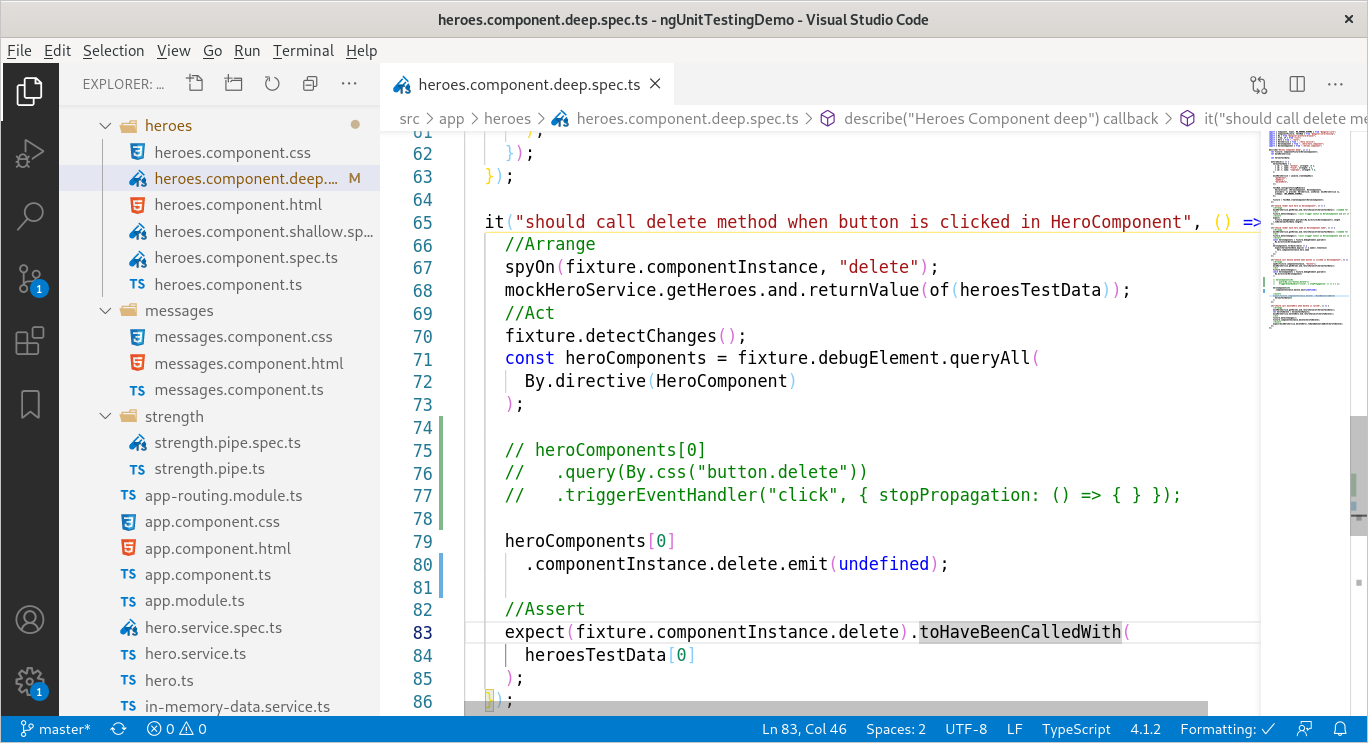
When we use inject, we want to preserve the orders of the parameters similar to the array passed to inject.

## Emitting events

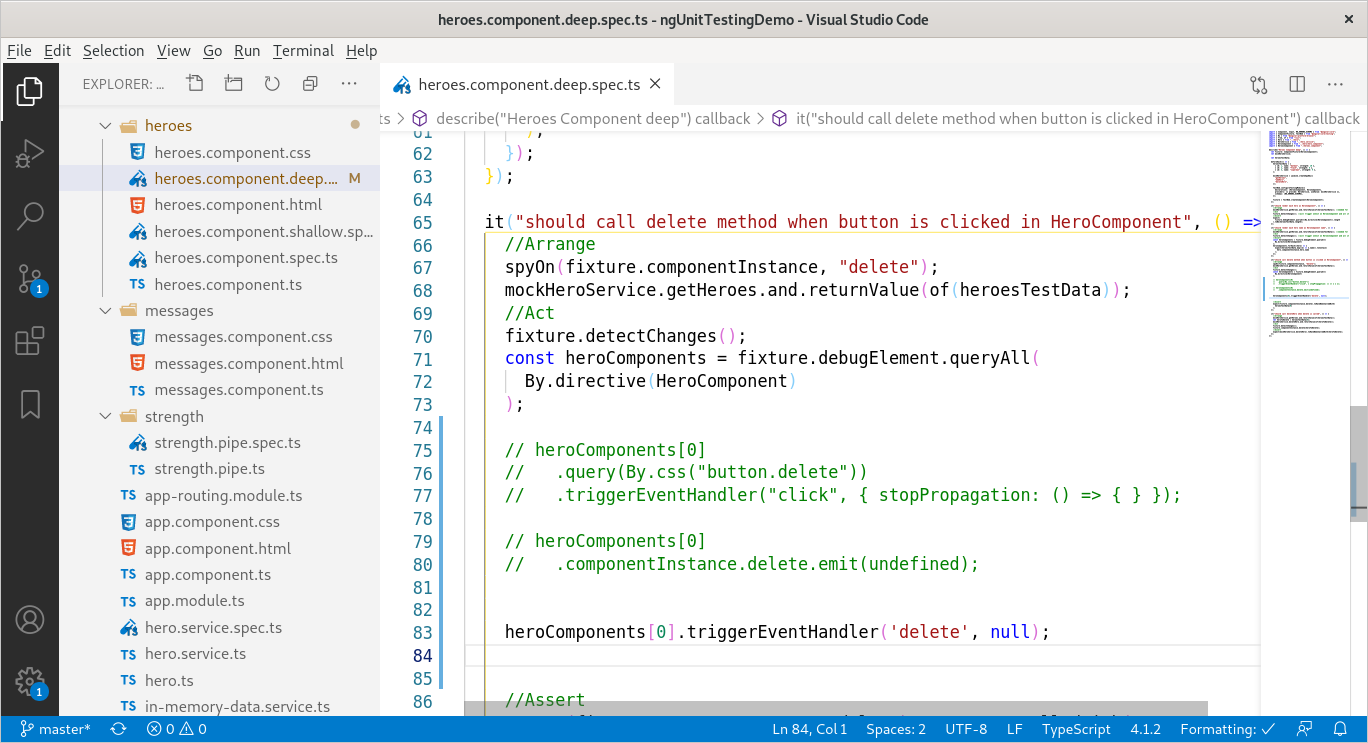
1. triggerEventHandler



1. Emit event by calling method

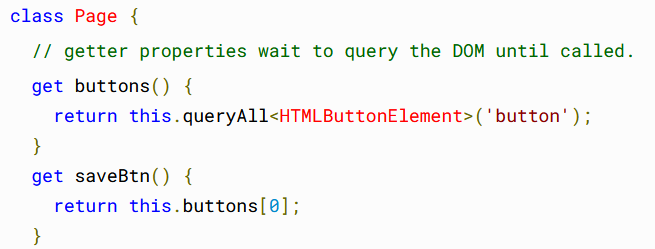


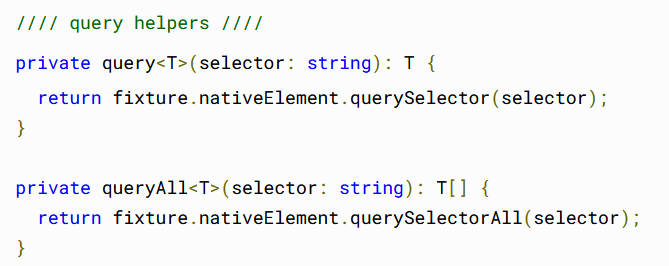
1. Trigger the event directly on the debugElement



## Using page object to test template

If the template is too complex (too much bindings) we can create a page class which acts like a service to query certain things in each page like save button, cancel button, and contains helper methods, instead of writing same logic in each separate component and test

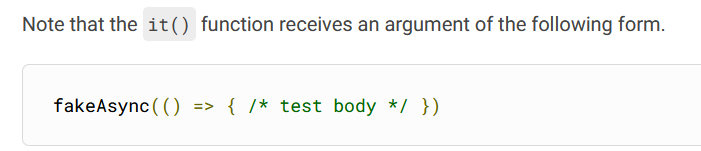




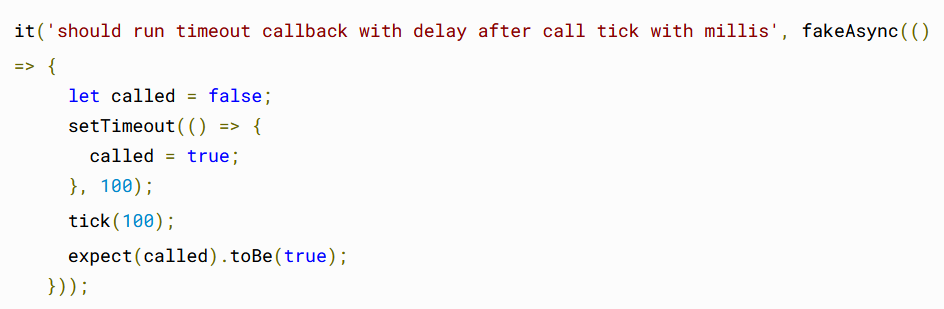
# Testing Async code

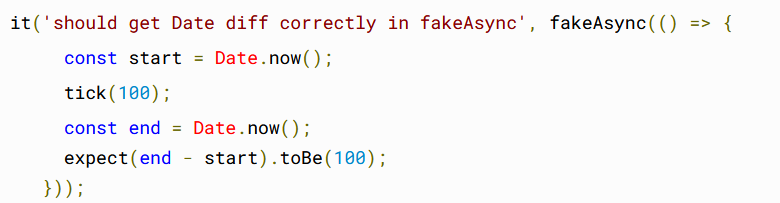
## Timeout

1. Wrap our test inside **fakeAsync** which returns a sync method as a test but waits for async code inside the test
   1. To use it we must use ‘zone.js/testing’

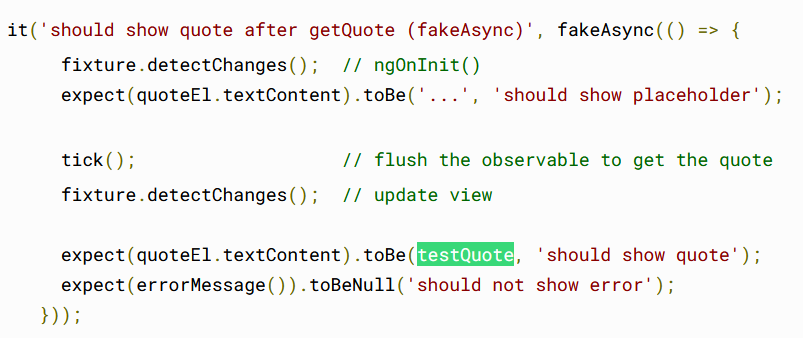


* 1. The [**fakeAsync**](https://angular.io/api/core/testing/fakeAsync)() function enables a linear coding style by running the test body in a special [fakeAsync](https://angular.io/api/core/testing/fakeAsync) test zone
  2. Limitation: The [**fakeAsync**](https://angular.io/api/core/testing/fakeAsync)() function won't work if the test body makes an **XMLHttpRequest** (XHR) call. XHR calls within a test are rare, but if you need to call XHR, see [**waitForAsync**()](https://angular.io/guide/testing-components-scenarios#waitForAsync),

1. Use **tick** function to fast forward the time we want to wait for async code
   1. You do have to call [**tick**()](https://angular.io/api/core/testing/tick) to advance the (virtual) clock.
   2. Calling [**tick**()](https://angular.io/api/core/testing/tick) simulates the passage of time until all pending asynchronous activities finish. In this case, it waits for the error handler's setTimeout().
   3. We can use tick to represent the passage of time



* 1. Tick flushes promises or observables and resolves them at once

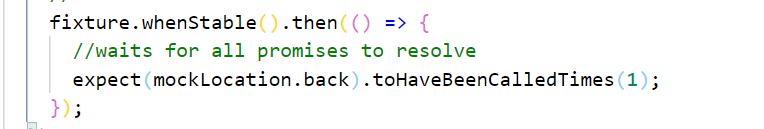


Behind the scenes testQoute gets set by an observable

1. We can use flush instead of tick when we don’t know the amount of time we need to wait

## Promises

1. Wrap the test inside async function
2. **fixture.whenStable()** which returns a promise that resolves when all other promises inside the component have resolved

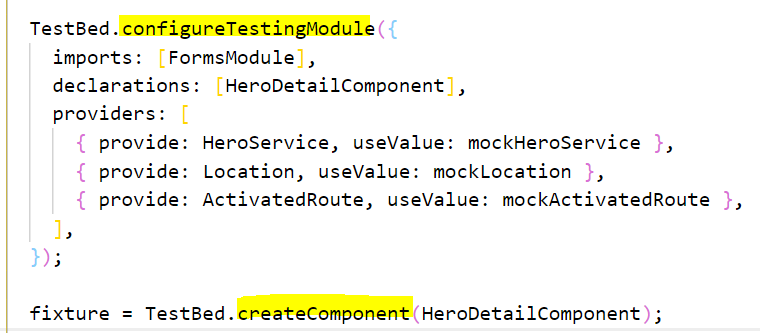


1. The fixture.whenStable() returns a promise that resolves when the JavaScript engine's task queue becomes empty

FakeAsync can work with both timeouts and promises while async and whenStable only work well with promises

# TestBed

We use it to create testing module and component fixture for testing and for resolving depencey injection



After we use createComponent we shouldn’t modify the testing module configuration anymore

* Do not re-configure [TestBed](https://angular.io/api/core/testing/TestBed) after calling createComponent.
* The createComponent method freezes the current [TestBed](https://angular.io/api/core/testing/TestBed) definition, closing it to further configuration.
* You cannot call any more [TestBed](https://angular.io/api/core/testing/TestBed) configuration methods, not configureTestingModule(), nor get(), nor any of the override... methods. If you try, [TestBed](https://angular.io/api/core/testing/TestBed) throws an error.

## Fixture

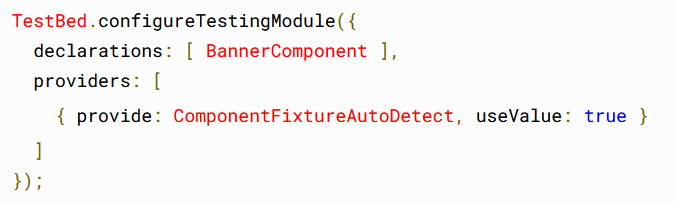
Fixture: its an object representing the component we are testing, but has few more properties than the component  
Contains 3 main objects:

* componentInstance: to manipulate the component object it self
* nativeElement: to access the component as dom element and use methods like getElementByClassName, querySelector,...etc
* debugElement: a wrapper for nativeElement

Change Detection

when we update properties on component instance, it doesn’t trigger the change detection in angular, so we need to do that manually through the test fixture

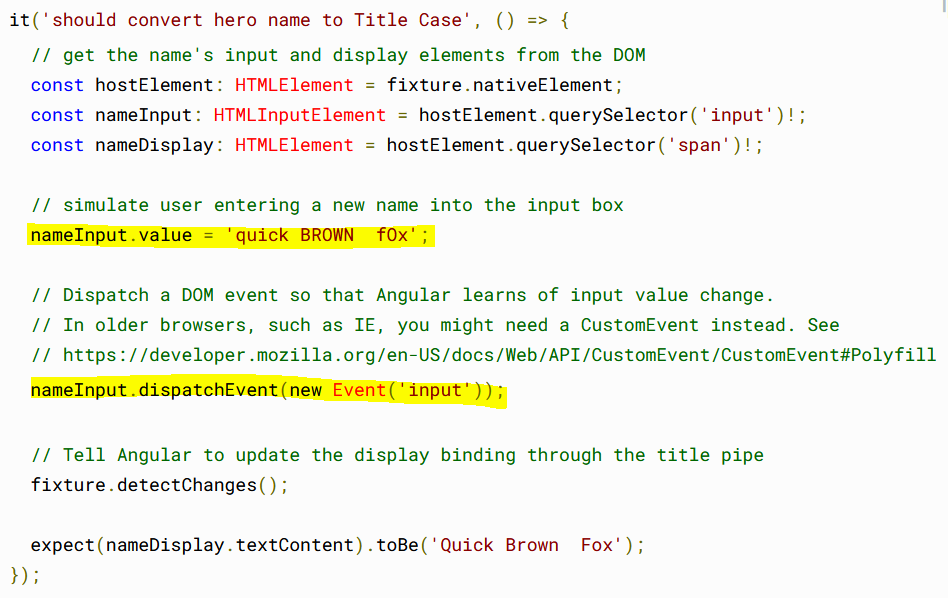
If we want angular to run change detection automatically we pass this configuration



Rather than wonder when the test fixture will or won't perform change detection, the samples in this guide always call detectChanges() explicitly. There is no harm in calling detectChanges() more often than is strictly necessary.

DispatchEvent

When we want to let angular know we made changes to input



NO\_ERRORS\_SCHEMA

when we are testing a component that has router links or form submissions, we don’t want the component to actually

* However it could suppress some other issues like wrong tags and incorrect html attributes

# Routing

## Testing routing in TS code

We can mock the router by creating a stub for ActivatedRouter to read params and query string passed in url

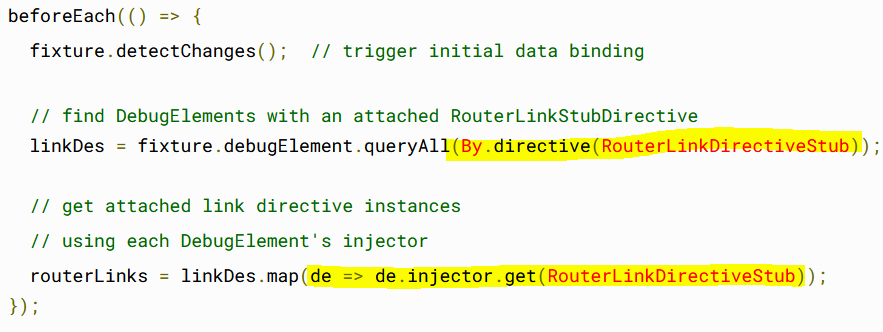
export class ActivatedRouteStub {}

The stub should contain mocking of objects like snapshot or params

## Testing nested components (route url)

We create stubs for the not needed components and directives, because we need to test the router-link url that it routes to a specific component





## Notes

* All of the test (complete story) (Arrange,Act,Assert) should be all inside the ‘it’ function
* If the code that is in the Arrange isnt critical for the particular test, then we can move it to beforeEach
* If same setup is critical for 2 tests then we can duplicate it in both tests
* Don’t test the framework, test how you are interacting with the framework

Jest

# Setup

Required packages:

* *jest* – Jest testing platform
* *jest-preset-angular* – configuration preset with common settings setup for you
* *@types/jest* – Jest typings

Add configurations to angular.config:

"test": "jest",

"test:watch": "jest --watch",

Add jest extension to vscode

# Matchers

* **ToBe**

Exact equality

Can be used to compare references as well

toBe uses Object.is to test exact equality

* **ToEqual**

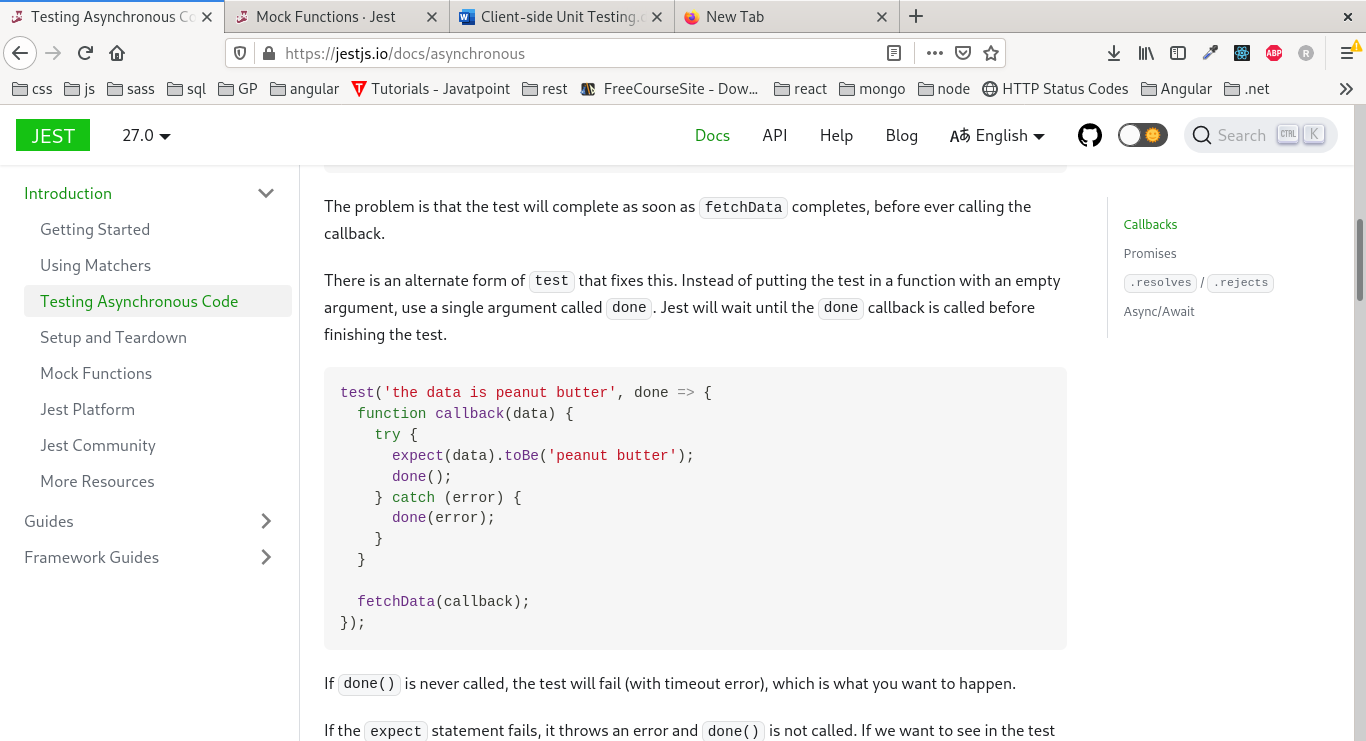
Used to compare objects values and arrays values

Recursivly checks object or array

<https://jestjs.io/docs/expect>

# Testing Async

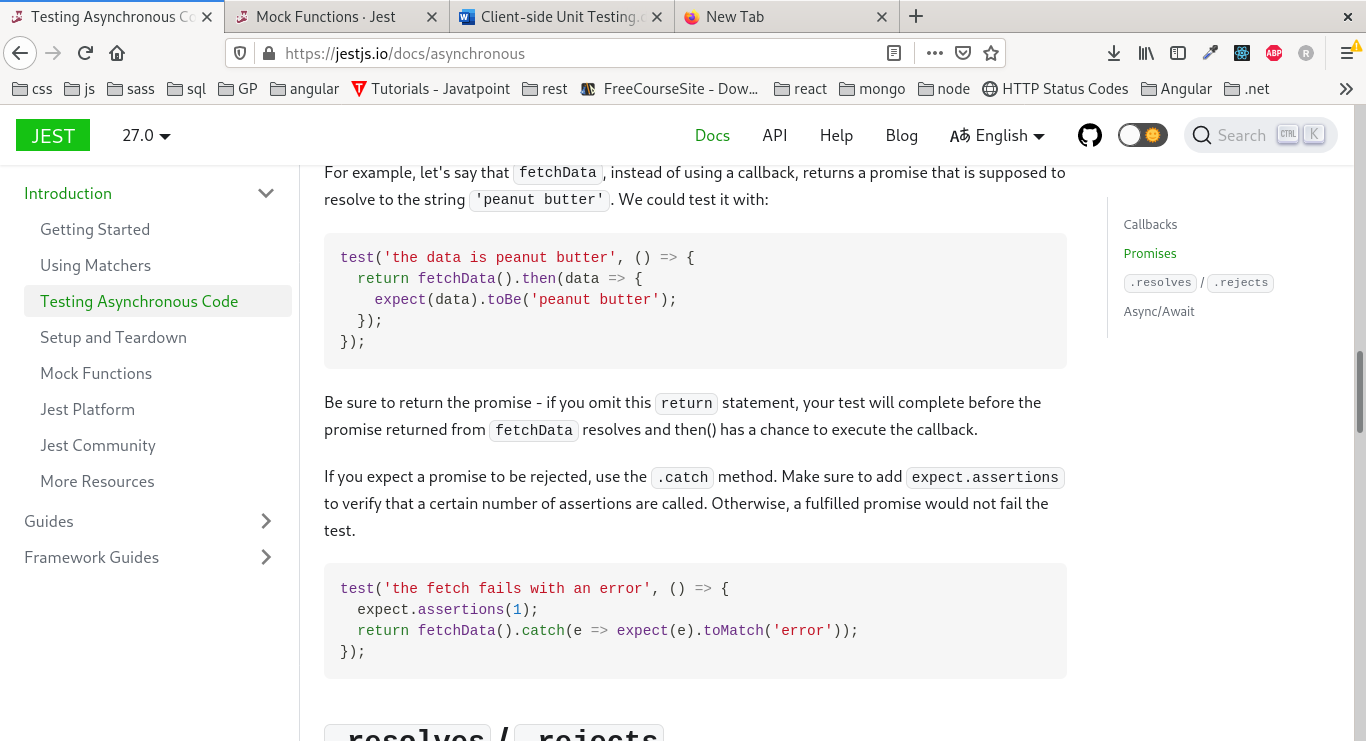
1. Using done function



If done() is never called, the test will fail (with timeout error),

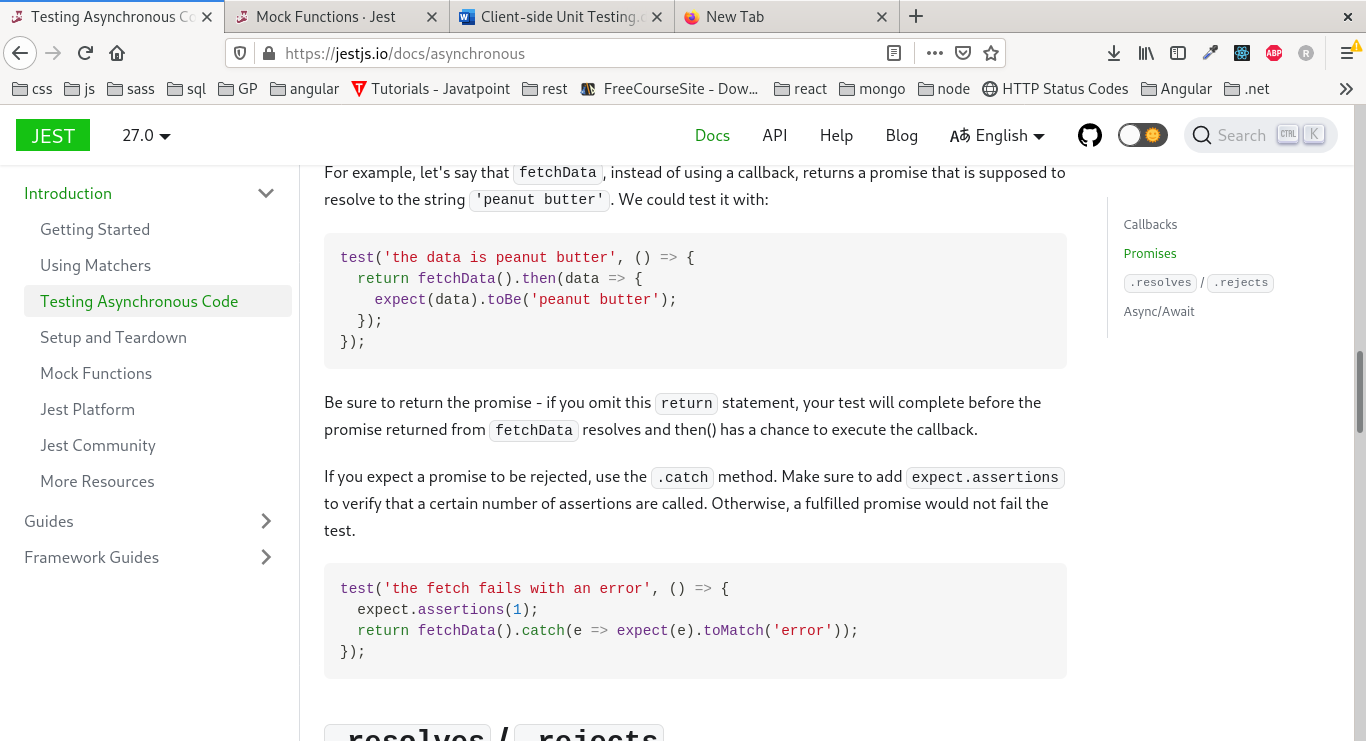
We wrap the test in try catch to see what error occured

1. Promises



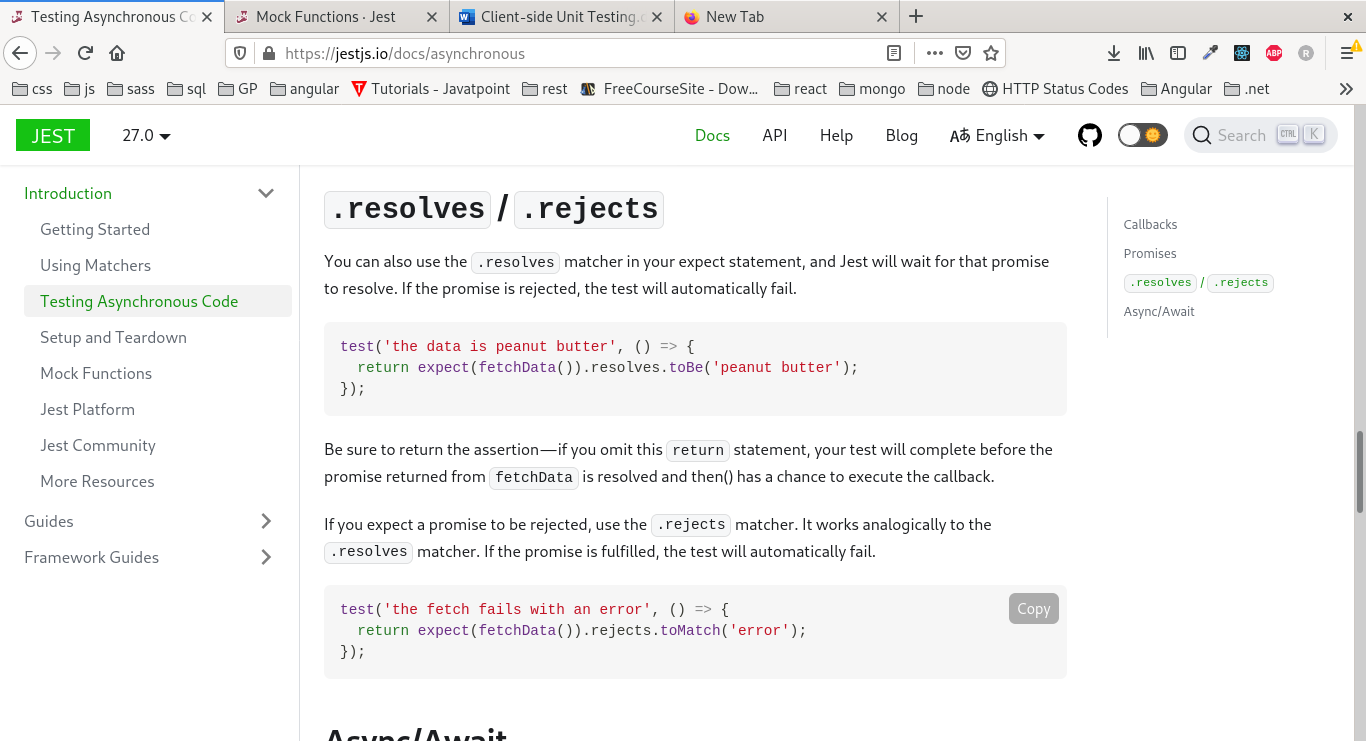
We must return the promise in order for the test to wait for it to be fullfilled

If we want to test a failing promise we can use catch function and assert it

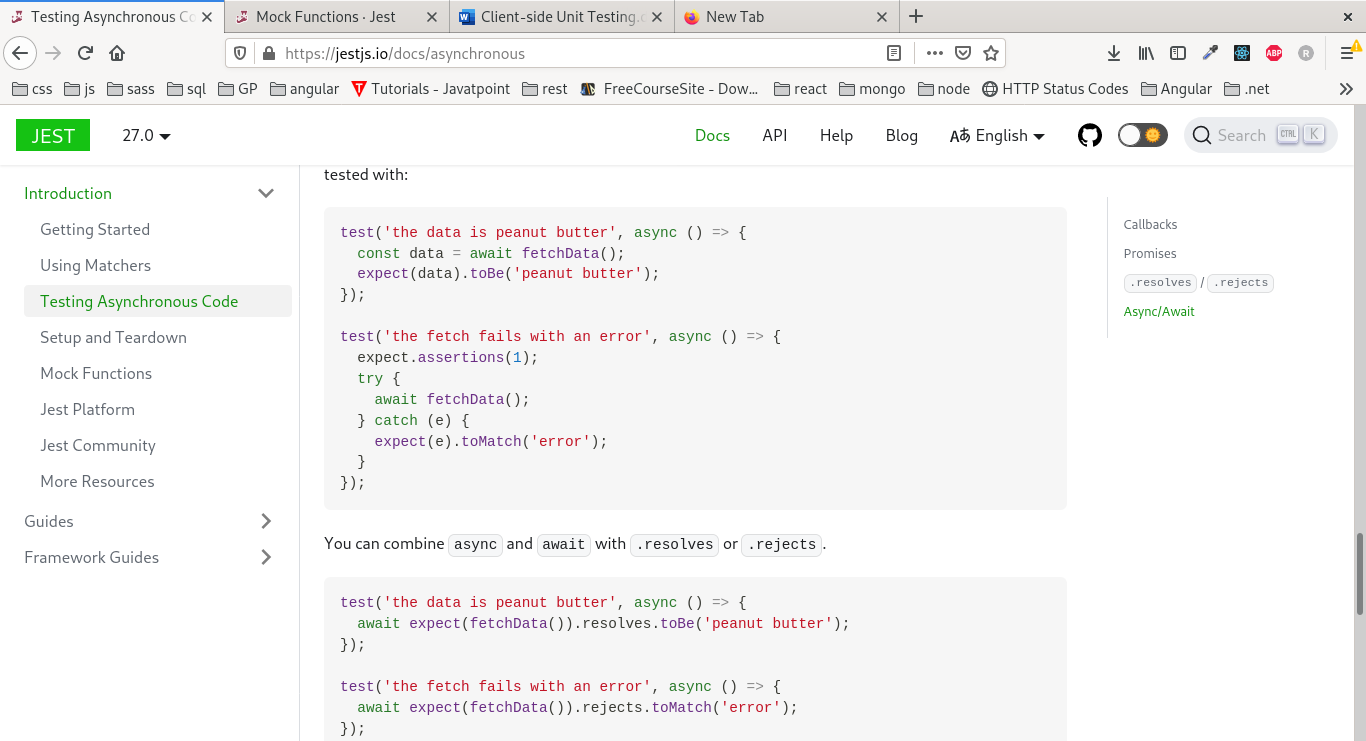


1. Resolves/Rejects

You can also use the .resolves matcher in your expect statement, and Jest will wait for that promise to resolve. If the promise is rejected, the test will automatically fail.



1. Async/Await



# Setup and teardown

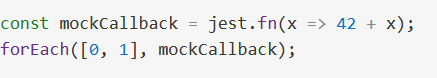


Its better to use these methods instead of writing code inside describe because it is always executed before any test, meaning all describe of all tests are executed before any test function execute

# Mocking

## Jest.Fn

We can use it to mock a callback and specify its implementation



In this example the function is the method we are testing

All mock functions have this special .mock property, which is where data about how the function has been called and what the function returned is kept.

*// The mock function is called twice*

expect(mockCallback.mock.calls.length).toBe(2);

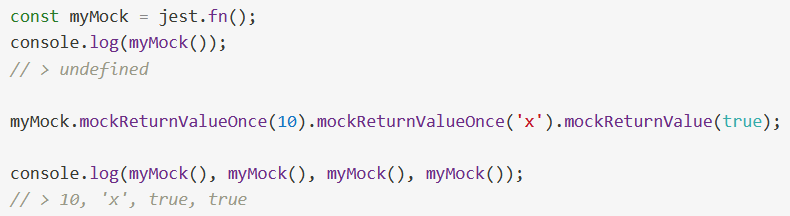
*// The first argument of the second call to the function was 1*

expect(mockCallback.mock.calls[1][0]).toBe(1);

*// The return value of the first call to the function was 42*

expect(mockCallback.mock.results[0].value).toBe(42);

we can also pass no argument to fn and mock the values it returns



## Mocking classes

We can use jest.mock to mock a module (exported class) which makes http calls for example and use mockResolvedValue to mock how the value is returned

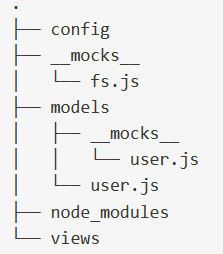


## Manual mocks

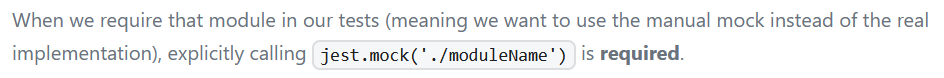
Manual mocks are used to stub out functionality with mock data.

For example, instead of accessing a remote resource like a website or a database, you might want to create a manual mock that allows you to use fake data.

Manual mocks are defined by writing a module in a \_\_mocks\_\_/ subdirectory immediately adjacent to the module.



For example, to mock a module called user in the models directory, create a file called user.js and put it in the models/\_\_mocks\_\_ directory



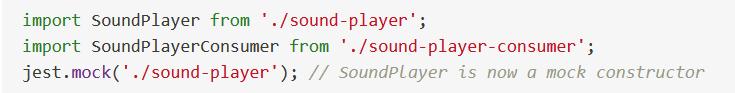
When a manual mock exists for a given module, Jest's module system will use that module when explicitly calling jest.mock('moduleName')

# ES6 Class Mocking

## Automatic mock

Calling **jest.mock('./sound-player'**) returns a useful "automatic mock" you can use to spy on calls to the class constructor and all of its methods

It replaces the ES6 class with a mock constructor, and replaces all of its methods with [mock functions](https://jestjs.io/docs/mock-functions) that always return **undefined**.



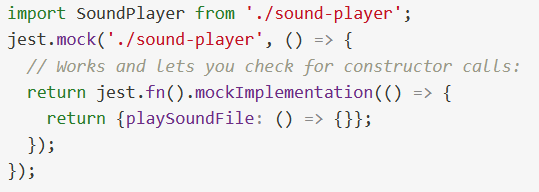
## Manual mocks

By adding stubs in \_\_mocks\_\_ folder

When we create manual mocks its hard to spy on constructor or methods

### Spying on constructor

In order to track calls to the constructor, replace the function returned by the HOF with a Jest mock function. Create it with [jest.fn()](https://jestjs.io/docs/jest-object#jestfnimplementation), and then specify its implementation with mockImplementation()



### Spying on methods

we can use spyOn to test calls to specific method inside a stub



## Module Factory parameter

jest.mock(path, moduleFactory) takes a **module factory** argument. A module factory is a function that returns the mock.

It must return the implementation either a function, constructor function, class.



A limitation with the factory parameter is that, since calls to jest.mock() are hoisted to the top of the file, it's not possible to first define a variable and then use it in the factory. An exception is made for variables that start with the word 'mock'. It's up to you to guarantee that they will be initialized on time! For example, the following will throw an out-of-scope error due to the use of 'fake' instead of 'mock' in the variable declaration

For example fakePlaySoundFile instead of mockPlaySoundFile

If we need to change the implementation of mock in one of our tests we can call mockImplementation()

You can specify a mock later, e.g. in beforeAll(), by calling mockImplementation() (or mockImplementationOnce()) on the existing mock instead of using the factory parameter. This also allows you to change the mock between tests, if needed

# References

Let’s start with Jasmine and karma.

<https://www.youtube.com/watch?v=yG4FH60fhUE>

<https://angular.io/guide/testing>

<https://medium.com/swlh/angular-unit-testing-jasmine-karma-step-by-step-e3376d110ab4>

<https://www.digitalocean.com/community/tutorials/angular-introduction-unit-testing>

This CONFERENCE is very useful, it's shown the different between (jest) and (karma and jasmine)

<https://app.pluralsight.com/library/courses/angular-denver-2019-session-03/table-of-contents>

<https://frontendmasters.com/courses/production-angular/testing/>

<https://app.pluralsight.com/guides/how-does-jest.fn()-work>

<https://app.pluralsight.com/guides/test-asynchronous-code-jest>

<https://jestjs.io/docs/es6-class-mocks>