Exercise 14 - Testing

Objective

To create tests for a Service and a Component in order to practice using Angular's testing tools.

Overview

In this exercise we are going to create two test suites. One for a service and another for a component - neither of which have been written yet. Initially we will plan the tests, write them, and check they fail, then we will write the code in order to pass the tests. Finally, once we have tested and passed both the service and component we will write a template to display a simple locations page.

Instructions

Part 1 - Testing the service

1. Start this exercise by creating a new `Locations` module and *importing it into the root* of the application. This is going to house (rather predictably) our location related features.
2. Create a new `LocationsBrowse` component and a **location service** within this module.
3. Using the CLI the spec files for both the component and the service have been generated for us - how handy! Run **ng test** in the command line to verify the tests run and pass.
4. It will be easier to write our service first, so let's start there. Before we write any actual service code we should write a test. This **service** will provide a function `getLocations()` which will make an *HTTP request* from our json-server. Using the existing `it()` statement that tests that the service is created as a template, add another so that it tests for the existence of the function `getLocations()`.

```
expect(service.getLocations).toEqual(jasmine.anything());
```

5. Run your test and watch it fail!
6. Now write the minimal amount of code to make this test pass (i.e. create the `getLocations` function)
7. Run your test and watch it pass!
8. Amend the test for the service to test that the service makes a HTTP request of the right type to the correct address by:
    - Importing `HttpTestingController` and `HttpClientTestingModule`

- Declaring *two method variables*, `service` as a `LocationsService` and `httpMock` as a `HttpTestingController` in the `describe` method for `LocationService`,
- Adding `HttpClientTestingModule` as `imports` to the `TestBed.configureTestingModule` object argument in the `beforeEach` method
- Assigning `service` to be a call to `TestBed.get()`, inserting `LocationService` *as an argument into it* in the `beforeEach` method
- Assigning `httpMock` to be a call to `TestBed.get()`, inserting `HttpTestingController` *as an argument into it* in the `beforeEach` method
- Defining a `QALocation` **model** that has properties of `location`, `mapUrl` and `mapSrc` and *importing* it into the **service** and the **test**

NOTE: This will be a new file defining a QALocation class.

- *Creating a second describe method* (inside the original one) that has `#getLocations` as a *string argument* and a *no-argument anonymous function* that:
  - Has an `it` method with a *string argument* of *should return an Observable<Location[]>*
  - `it` has a *second no-argument anonymous function argument* that:
    - Declares a **const** `dummyLocations` of type `Location[]` *assigned to an array of 2 objects* that have the *requisite properties for a Location object*
    - Subscribes to the `getLocations` method from the `LocationService` and has an argument of `locations` expecting that `locations` `toEqual` the `dummyLocations`

```
service.getLocations.subscribe(locations => {
    expect(locations).toEqual(dummyLocations);
}
```

    - Declares a **const** `req` and assigns this to

```
httpMock.expectOne(`http://localhost:3000/locationsResults`)
```

    - Expects `req.request`'s `method` `toBe` `GET`
    - Calls `flush` on `req` *with an argument of* `dummyLocations`

9. Run the test and it should fail in the command-line/terminal before you even get to see the results from Karma.

- This is because the getLocations method is not returning an object of type Observable

10. The `getLocations` method in the **location.service.ts** file should return the result of a **HttpClient** call to the *json-server* for **locationsResults** (make sure that you import `HttpClient` and *inject it into the constructor*)
11. Save all of the files and run the test again. It should pass if you've done everything correctly

- If you get errors, try restarting the test in the command-line/terminal

Part 2 - Testing the component

Let's write the tests for our component. The component is going to be very simple: it will request location data from the location service and populate a property once the data is retrieved.

12. Start by adding a new `it()` statement and articulate that this component should request data from the service
13. We're going to stub the service. Create *two new variables at the top of the describe block*: `locationServiceSpy` and `locationService`.
14. Add a **const** `DUMMYLOCATIONS` of type `QALocation[]`
    a. Populate this array with 3 dummy objects containing the requisite properties for a **QALocation** object instance

The Spy is going to stub for the service, and then we will provide this spy to the test as locationService - which gives us control and feedback as to what the component is trying to do with the service.

15. We know the **location-browse** component should request the locations so *at the top of the first* `beforeEach` block *create a new object* assigned to the variable `locationServiceSpy` and give it a `getLocations` property which is a **jasmine spy**:

```
Jasmine.createSpy('getLocations').and.callFake(callback)
```

16. Our callback is going to be *a function that returns an Observable* which *emits the array of locations* (`DUMMYLOCATIONS`)

```
() => { Observable.create(observer=>{observer.next(DUMMYLOCATIONS)}) }
```

17. Now we need to provide this stub of the service. Add a `providers` array in the `TestBed.configureTestingModule` block - it needs to `provide` the expected `LocationService` but actually giving the `locationServiceSpy` as a `value`

```
providers: [
```

```
  { provide: LocationService, useValue: locationServiceSpy }
```

18. *At the end of the second* `beforeEach` function, add the following line to inject the `locationService` into the component created for testing:

```
locationService = fixture.debugElement.injector.get(LocationService)
as any;
```

19. Now to write the actual test. Within the `it()` statement you created earlier add the following lines of code:

```
it('should call getLocations', () => {
        expect(locationService.getLocations.calls.count()).toBe(1,
'getLocations called');
})
```

20. This checks with our Spy whether `getLocations` has been called. **Run the test and confirm that this test fails**
21. Let's now write the minimal amount of code to get this test to pass
22. *Inject the service* into the component and call its `getLocations()` function from within the `ngOnInit` function
23. Re-run the tests and you should now find it passes
24. The locations-browse component should set a property `locations` to be that of *the array emitted by the Observable*. Write a suitable test for this using the following code as a guide

```
expect(component.locations).toEqual(array);
```

25. Now this won't compile because `locations` doesn't exist on the component. **Create it** and re-run the tests. It should now fail because locations is *undefined*
26. Update your component so that it *subscribes* to the **return** of `getLocations()` and sets the `locations` property of the component properly to pass the test

If you have time...

27. Create a template for the locations module so that you can display the location and map
28. Create a suitable route so that you can navigate to this component and add the necessary attributes to hook up the link in the navigation bar

Note: Due to the limits on using the google API without an API key the images may or may not load. You can check the console to ensure the correct requests were made - you may find 403 (Unauthorised) responses and if you really want the site to work perfectly, substitute mapUrl for mapSrc in the code to use the pre-saved images!

If you really, really have time…

29. Write a test to ensure that the Instructor Service is working correctly
30. Write a test to ensure that the Course Service is working correctly

- You may find the following useful:

Testing a request:

```
expect(requestName.request.method).toBe('<RequestType>');
```

Testing the request body:

```
expect(requestName.request.body).toBe(<Some data to test against>);
```