Angular
Lab 10
Testing Angular Apps

1 LAB SETUP **1**

2 GENERATING A NEW ANGULAR PROJECT **2**

3 BASIC UNIT TESTS WITH JASMINE **3**

4 TESTING COMPONENTS **5**

4.1 TESTING CHILD COMPONENTS 7

5 TESTING DIRECTIVES **8**

6 TESTING SERVICES **8**

6.1 TESTING COMPONENTS THAT USE SERVICES 9
6.2 TESTING SERVICES THROUGH THE USE OF MOCKS 9
6.3 TESTING SERVICES THAT USE HTTPCLIENT DIRECTLY AND VIA A MOCK 10

1 Lab setup

Make sure you have the following items installed

- Latest version of Node and NPM (note: labs are tested with Node v16.13 and NPM v8.1 but should work with later versions with no or minimal changes)
- Latest version of TypeScript (note: labs are tested with TypeScript v4.5.4 but should work with later versions with no or minimal changes)
- Latest version of Angular (note: labs are tested with Angular v13.1 but should work with later versions with no or minimal changes)
- Visual Studio Code (or a suitable alternative IDE for Angular/TypeScript)
- A suitable text editor (Notepad ++)
- A utility to extract zip files (7-zip)

In each of the main lab folders, there are two subfolders: `changes` and `final`.

- The `changes` subfolder holds the source code and other related files for the lab. The creation of the Angular app will proceed in a step-wise fashion in order to clearly demonstrate the various features of the framework. The various project source files in this folder are numbered in a way to reflect this gradual construction. For e.g. `xxx.v1.html`, `xxx.v2.html`, etc represents successive changes that are to be made to the `xxxx.html`

- The `final` subfolder holds the complete Angular project. If there was some issue in completing the step-wise construction of the app using the files from the `changes` subfolder, you can use this to run the final working version.

The project folder in `final` is however missing the crucial `node_modules` subfolder which contains all the dependencies (JavaScript libraries) that your Angular app needs in order to be built properly. These dependencies are specified in the `package.json` file in the root project folder. They are omitted in the project commit to GitHub because the number and size of the files are prohibitively large.

In order to run the app via the Angular development server (`ng serve`), you will need to create this subfolder containing the required dependencies. There are two ways to accomplish this:

a) Copy the project root folder to a suitable location on your local drive and run `npm install` in a shell prompt in the root project folder. NPM will then create the `node_modules` subfolder and download the required dependencies as specified in `package.json` to populate it. This process can take quite a while to complete (depending on your broadband connection speed)

b) Alternatively, you can reuse the dependencies of an existing complete project as most of the projects for our workshop use the same dependencies. In this case, copy the `node_modules` subfolder from any existing project, and you should be able to immediately start the app. If you encounter any errors, this probably means that there are some missing dependencies or Angular version mismatch, in which case you will then have to resort to `npm install`.

The lab instructions here assume you are using the Chrome browser. If you are using a different browser (Edge, Firefox), do a Google search to find the equivalent functionality.

2 Generating a new Angular project

In either a separate command prompt (or shell terminal) or in an embedded terminal in Visual Studio Code, create a new Angular project with:

```
ng new testingdemo
```

Press enter to accept the default values for all the question prompts that follow regarding routing and stylesheet.

Navigate into the root project folder from the terminal with:

```
cd testingdemo
```

Build the app and serve it via Angular's live development server by typing:

```
ng serve
```

The final output line from this should indicate that the compilation process was successful and identify the port that the live development server is currently serving up the Angular app dynamically from (by default this will be port 4200)

Open a browser tab at:
http://localhost:4200/

You should be able to see the default landing page for all new autogenerated Angular projects.

To stop the development server, type Ctrl+C in the terminal window that it is running in. You can restart again anytime by typing `ng serve` in the project root folder.

3 Basic unit tests with Jasmine

Angular uses the Jasmine framework for creating test specs and the Karma test runner for running the specs. All of these are preinstalled with the Angular framework (including test reporters, browser launchers and misc tools), so we can start using them directly without the need for any installation steps.

The source code for this lab can be found in the `changes` subfolder of `Testing-Demo`

Modify / add the following files in `src/app` with the latest update from `changes`:

`app.component.spec-v1.ts`

`person-v1.ts`

There is a corresponding Jasmine spec file for every component in an Angular project ending with a matching first portion of the name but ending with `*.spec.ts`

Key methods in this spec file:

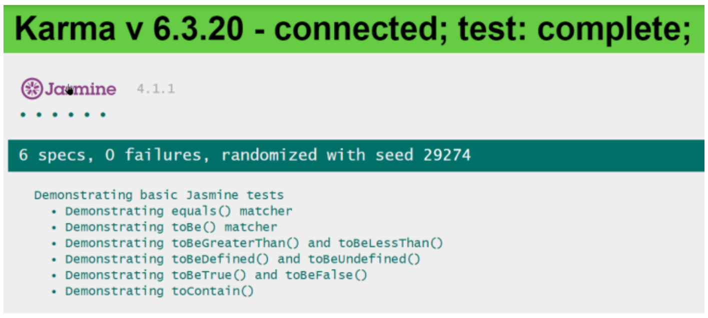
- `describe()` – declares a suite, which contains one or more tests (or specs). Each suite describes a piece of code, the code under test. This function takes two parameters: a descriptive string and a function (typically arrow function) containing the suite definition. It also typically contains a beforeEach block.
- `it()` – This is the spec. It's the smallest unit test case that is written to be executed, which calls a global Jasmine function with two parameters: a descriptive string and a function containing one more expectations - `expect()`. Typically, there are multiple specs within the main `describe()`.
- `expect()` – Every `it()` statement has a `expect()` function which takes a value called an actual. It is used alongside a matcher function. Matcher functions take a value that represents the expected value and is chained to an `expect()` function. Together they return a boolean value that indicates whether the spec passes or fails.

In a command prompt of the root folder of this project, run the single test spec (`app.component.spec.ts`) with:

```
ng test
```

This command builds the application in watch mode, and launches the Karma test runner. The rest runner executes all the tests in all the spec files (`*.spec.ts)` in the Angular projects. Any output reporting the test specs being run and their results, as well as any `console.log` statements is directed to the console. Additional test output is also available as a service (the Jasmine HTML reporter) which is by default at `localhost:9876` (a browser tab will automatically be opened at this URL, otherwise you can explicitly type this URL yourself). The `console.log` statements will be visible in the Console tab of the Chrome Dev tools at this tab.

Occasionally after making changes to the spec files and saving, the changes are not picked up by the test runner and the tests are not rerun accordingly. In that case, you will then have to stop the test runner and start it again with `ng test`.



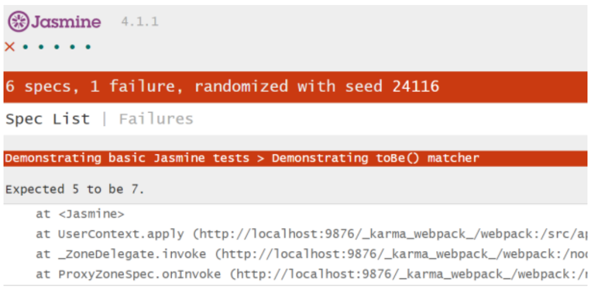
```
Karma v 6.3.20 - connected; test: complete;
Jasmine 4.1.1
• • • • • •
6 specs, 0 failures, randomized with seed 29274
  Demonstrating basic Jasmine tests
    • Demonstrating equals() matcher
    • Demonstrating toBe() matcher
    • Demonstrating toBeGreaterThan() and toBeLessThan()
    • Demonstrating toBeDefined() and toBeUndefined()
    • Demonstrating toBeTrue() and toBeFalse()
    • Demonstrating toContain()
```

Click the top-level link for the suite description (`Demonstrating basic Jasmine tests`): this reruns all the test specs in the test suite. Notice that the sequence of execution of these test specs is random each time the test suite is rerun.
You can also click on the individual spec descriptions (in bullet points) to run only that spec.

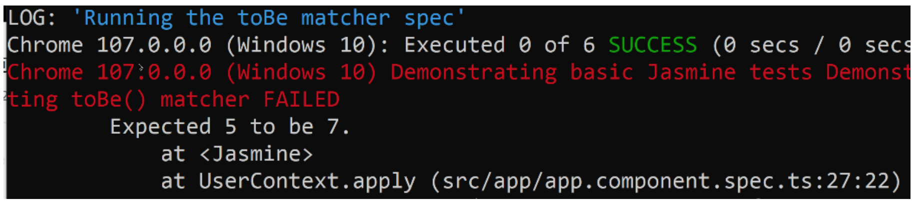
You can make modifications to the various spec files while the test runner is still active in watch mode: it will pick up these changes and reruns the suite again.
Make some modifications to any of the matchers in any of the specs in `app.component.spec.ts` that will cause one or more of the specs fail, for e.g. in `Demonstrating toBe() matcher`

```
expect(firstNum).toBe(7);
```

Notice that the failure in the spec is flagged in red in the browser as well as console output.



```
Jasmine 4.1.1
×• • • • •
6 specs, 1 failure, randomized with seed 24116
Spec List | Failures
Demonstrating basic Jasmine tests > Demonstrating toBe() matcher
Expected 5 to be 7.
    at <Jasmine>
    at UserContext.apply (http://localhost:9876/_karma_webpack_/webpack:/src/a
    at _ZoneDelegate.invoke (http://localhost:9876/_karma_webpack_/webpack:/no
    at ProxyZoneSpec.onInvoke (http://localhost:9876/_karma_webpack_/webpack:/
```



```
LOG: 'Running the toBe matcher spec'
Chrome 107.0.0.0 (Windows 10): Executed 0 of 6 SUCCESS (0 secs / 0 secs
Chrome 107.0.0.0 (Windows 10) Demonstrating basic Jasmine tests Demonst
ting toBe() matcher FAILED
        Expected 5 to be 7.
            at <Jasmine>
            at UserContext.apply (src/app/app.component.spec.ts:27:22)
```

Inside the `it()` block lies the actual testing code. These usually consists of 3 distinct phases. Arrange, Act and Assert.

a) Arrange is the preparation and setup phase. When we start working with Angular classes in the upcoming lab sessions, this will usually require that the class under test is instantiated. Other activities can include setting up dependencies, spies and fakes.
b) Act is the phase where interaction with the code under test happens. For example, a method is called or an HTML element in the DOM is clicked.
c) Assert is the phase where the code behavior is checked and verified. For example, the actual output is compared to the expected output.

When writing multiple specs in one suite, the Arrange phase is similar or even identical across these specs. For example, in most Angular unit tests, the Arrange phase always consists of creating a Component instance and rendering it into the document. Since this set up is repeated multiple times, to avoid unnecessary code redundancy, we can define it once in a central setup function. This function is then called prior to the beginning of each spec. Based on that idea / principle, we can also set up code that is called before and after each spec, or before and after all specs. This is achieved using the functions: `beforeEach`, `afterEach`, `beforeAll` and `afterAll`.

`app.component.spec-v1-2.ts`

Verify that the sequence of console log statements confirms with the purpose of each of those functions.

References:

https://testing-angular.com/test-suites-with-jasmine/#test-suites-with-jasmine

https://jasmine.github.io/api/edge/matchers.html

4 Testing components

Modify the following files in `src/app` with the latest update from changes:

`app.component-v2.ts`

`app.component-v2.html`

`app.component.spec-v2.ts`

Make sure you have the Angular CLI development server (`ng serve`) and the Karma test runner (`ng test`) running in two separate terminals.

Test out the basic functionality implemented by interpolation and event binding in the app, before examining the test spec implementation.

The TestBed creates and configures an Angular environment so you can test particular application parts such as components and services safely and easily. This comes with a testing Module that is configured in a similar manner to the root module (@NgModule) which supports the declarations of components, directives, pipes, services, etc. It has a static method `configureTestingModule` that allows this configuration.

The configuration code is typically placed in a `beforeEach` block because it needs to be repeated before every spec. This is usually inside an async-await because this operation proceeds in an asychronous manner and we want to ensure that it completes before we use it in each respective it() spec.

The createComponent returns a ComponentFixture, which is essentially a wrapper around the Component with useful testing tools. In the context of Angular, the ComponentFixture holds the Component and provides a convenient interface to both the Component instance and the rendered DOM. Via this fixture, we can access the component instance and through that we can access public properties in the component that we can test for initial values / updated values after an Angular change detection.

We can also use `fixture.debugElement.nativeElement` to access the top level document object for the DOM. We can then invoke a typical querySelector or querySelectorAll method using the standard CSS selectors to locate a specific element in the DOM, and then access specific properties in that element for the purposes of testing. Several examples of usage of CSS selectors are shown here.

In the event that you have multiple identical elements in the DOM, you can use either the id attribute or data-* attribute in order to uniquely identify that element for the purpose of referencing it for test purposes. Typically, you will use the data-* attribute since the id attribute can be used for other non-testing, application-specific purposes.
You can also use the class attribute, however, typically this attribute is used to specify class names for multiple elements, so you will then need to use the querySelectorAll method to obtain a NodeList representing a list of document elements that match that class and then provide additional logic to identify the element to be tested.

There are generally 3 kinds of tests that you would perform a component level:

a) Check that a particular component property has a specific value
b) Check that a particular template DOM element has a particular content (for e.g. a text-based element like <p>, <h1>-<h6> contains certain text)
c) Check that a particular template DOM element has a particular property (for e.g. the <a> tag has a `href` attribute with a specific value).

The checks in the 3 above situations are performed either when the component / template DOM have just been initialized or after a specific event has occurred in the template that might result in their values being updated.

In the testing environment, Angular does not automatically pick up events on various input elements or perform change detection and update the template DOM correspondingly. This must be done manually in the respective tests that require such actions.

a) Manually trigger an event (input, click, etc) on an event (for e.g. `fnameNode.dispatchEvent(new Event("input"));`)
b) Manually trigger the change detection with `fixture.detectChanges();`

One (or both) of these actions must be performed in order to ensure that all bindings (interpolation, property binding, two-ways binding, etc) is updated accordingly. Finally, you can perform the actual test using the `expect()` and `matcher` functions.

Modify the following files in `src/app` with the latest update from changes:

```
app.component.spec-v2-2.ts
```

There are certain variables / constants that are reused in some / all of the test specs, so we can streamline our test implementation by moving their declaration to the start of the declaration block and initialize them in the `beforeEach` block.

References:

https://angular.io/guide/testing-components-basics#nativeelement
https://angular.io/guide/testing-components-basics#debugelement

4.1 Testing child components

Open a new command prompt (separate from the ones that you are running `ng serve` in) in the root folder of the project created earlier. Type this command to generate a new component:

```
ng generate component firstChild
```

Modify the following files in `src/app` with the latest update from `changes`:

```
app.component-v3.ts

app.component-v3.html

app.component.spec-v3.ts
```

Modify the following files in `src/app/first-child` with the latest update from `changes`:

```
first-child.component-v3.ts

first-child.component-v3.html
```

Make sure you have the Angular CLI development server (`ng serve`) and the Karma test runner (`ng test`) running in two separate terminals.

Verify that the first attempt to perform a direct integration testing of the property binding between parent and child component fails.

To perform a unit test verifying the correct binding between the parent and child component in the child selector tag, we need to use a mock. There a few mocking frameworks available for Angular, a popular one is ng-mocks.

To install `ng-mocks`, type into the root folder of the project in a new command prompt:

```
npm install ng-mocks --save-dev
```

Typically, for testing related packages, we install them using the `--save-dev` option so that they will not be included in the final build meant for production deployment.

Modify the following files in `src/app` with the latest update from `changes`:

`app.component.spec-v4.ts`

Verify now that using the mock, we can test the correct binding between the parent and child component as well as ensuring that the parent property changes in accordance to an event emitted from the child component.

Modify the following files in `src/app/first-child` with the latest update from `changes`:

`first-child.component-v5.spec.ts`

When checking whether an @Output property (of EventEmitter<T> type) changes value as a result of an emit method called on it in a component method, we have two possible approaches:

- using a spy to check the emit method was called on the @Output property
- subscribing to the @Output property and executing a callback in response to a change its value (this is possible since EventEmitter is a subclass of Subject that extends Observable)

5 Testing directives

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v6.ts`

`app.component-v6.html`

`app.component-v6.css`

`app.component.spec-v6.ts`

Here we are testing some of the attribute and structural directives available. All of these affect the template directly, so testing will again involve triggering specific events on the template and verifying that the directives involving those events affect the change correctly.

6 Testing services

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v7.ts`

`app.component-v7.html`

`app.component.spec-v7.ts`

`basic.service-v7.ts`

`basic.service.spec-v7.ts`

Here, we implement a simple method in the service class that locates the longest string in an array of strings. The service is injected into the root component in the usual manner and a simple demo is provided.

We can test the service directly (via its corresponding `basic.service.spec.ts`) in a unit test independently of the component that it is injected into. We have to remember to first inject the service explicitly into TestBed so that it can be configured via the standard Angular DI mechanism discussed in an earlier lab. We perform 3 tests on the single method in the service: the first is a test with a typical array to test expected functionality, and the remaining 2 tests are testing boundary conditions (array with a single element and an empty array).

References:

https://angular.io/guide/testing-services#testing-services

6.1 Testing components that use services

Modify the following files in `src/app` with the latest update from `changes`:

`app.component.spec-v8.ts`

This is a form of integration testing, where we test AppComponent with the required service injected into it. To do this, we have to ensure that BasicService is registered explicitly in the `providers` property along with the declaration of AppComponent in the configureTestingModule call to TestBed.

We test that the component utilizes the service correctly by providing a new string array and verifying that the longest value is returned and rendered correctly to the template.

6.2 Testing services through the use of mocks

Modify the following files in `src/app` with the latest update from `changes`:

`randomstrings.service-v9.ts`

`app.component-v9.ts`

Here we introduce an additional new service RandomStringsService with a single method `getRandomArrayStrings` that returns an array of random length containing strings of random length. This service is utilized by AppComponent to initialize its `strArray` property. Verify that the content of this array is always random every time the app is reloaded, with the result that the longest string in the array is always changing.

Notice now the test in `app.component.spec.ts` is now failing because the `longestString` property is now being set based on the value returned from the method `getRandomArrayStrings`, which will be a random array of strings every time it is called. One way to get the test to work again is to also explicitly set the value of this property in the test suite, but this means we are duplicating nearly the entire logic of the component in the test suite, which is less than ideal.

The logical and better alternative is to use a mock as a spy object in place of the actual `RandomStringsService` service, so that when the `getRandomArrayStrings` is called on it, the spy can be set up to return a fixed dummy array of strings. This allows the final verification of the <p> element in the template to always have a fixed value as well.

Modify the following files in `src/app` with the latest update from `changes`:

`app.component.spec-v9.ts`

Notice now that the test now succeeds because the injection of the RandomStringsService into the component is now replaced with the spy, which will always return a fixed array of strings in response to the call to `getRandomArrayStrings` (unlike the actual `RandomStringService`). This guarantees that calling `findLongestString` on this fixed array of strings will always return a fixed value, which makes it easy to write the test correctly and verify the correct functionality of both the component as well as `BasicService`.

References:

https://angular.io/guide/testing-services#angular-testbed

6.3 Testing services that use HTTPClient directly and via a mock

Modify the following files in `src/app` with the latest update from `changes`:

`app.component-v10.ts`

`app.component-v10.html`

`app.module-v10.ts`

`fakeAPI.service-v10.ts`

`post-v10.ts`

`fakeAPI.service.spec-v10.ts`

`app.component.spec-v10.ts`

Here, we are using a simplified implementation of a service to make HTTP requests to the fake API service (https://jsonplaceholder.typicode.com) that we completed in a previous lab.

Verify that the service is working as expected by making a request to post id of 5.
Verify as well as the new single test spec in `fakeAPI.service.spec.ts` completes successfully as well. We have replaced the previous test specs in `app.component.spec.ts` with a dummy test spec will always succeed since we want to focus here on testing the service making the HTTP requests.

Note the key points in the set up of the test in `fakeAPI.service.spec.ts`

- In the configuration of TestingModule for the Testbed, we ensure that we include HttpClientModule in the `imports` property and the FakeAPIService in the `providers` property. We also explicitly inject HttpClient and FakeAPIService into the TestBed. This replicates the settings in the root module (`app.module.ts`) for the case of a normal component / service, which now needs to performed explicitly here for the case for a test spec
- In the single test spec that tests the `getSinglePost` method from `fakeAPIService`, we need to include the `done:DoneFn` to ensure that the subscribe method call on the observable returned from the `getSinglePost` method completes successfully.

Modify the following files in `src/app` with the latest update from `changes`:

`fakeAPI.service.spec-v11.ts`

Testing that involves making real HTTP calls to a backend server can be time consuming if the targeted server is slow to respond and / or the bandwidth of the connection is limited. To resolve, this Angular provides way to mock the HTTPClient (using HttpTestingController), so that all calls made by any service to HTTPClient is intercepted by the mock instead and an immediate predefined response can be returned without any actual HTTP calls being made. In this way, the tests can focus on the logic implemented by the service class that initiates the HTTP calls and / or the logic of any other components that use that service.

The tests involving the use of the mock httpTestingController involves expectations that certain requests have been made (i.e. a specific HTTP method such as GET, POST, PUT, etc) and which URL the request was made to. At the end, tests can also verify that the app made no unexpected requests.

References:

https://angular.io/guide/testing-services#testing-services
https://angular.io/guide/http#testing-http-requests