Testing

BUILDING WEB APPLICATIONS USING ANGULAR

Contents

- What are Unit Tests, TDD, BDD
- Recommended testing tools
  - Jasmine
  - Karma
- Testing in Angular
- Testing Angular Components
  - Using the TestBed
  - Change detection
  - Compiling the component
- Testing HTTP requests and Services

Introduction

- Testing is an integral part of development. It is a design tool not a debugging tool.
- Testing is not unique to Angular, although Angular does provide helpful utilities to aid with testing

Unit tests, BDD, TDD

- Unit tests focus on a single ‘unit’ of code – an object or module that we wish to test in isolation. That is – no dependencies on things like network access or databases
  - Unit tests needn’t be written in any special language or syntax – you *could* write them in plain JavaScript if you wish. But often we use Mocha, Jasmine and other tools as they make life easier and provide helpful features
- Test Driven Development (TDD) is a process whereby we write our tests, ensure they fail, and only then do we write the minimum amount of code to pass our test. The benefit of this approach is a very high test coverage (usually 90-100%)
- Behaviour Driven Development (BDD) is a set of practices on how to test our code. This can and should be used in combination with TDD and Unit testing methods. BDD is about testing behaviours rather than implementation

Testing in Angular

- There are various tools and techniques we can use to write Angular tests, we are going to use those recommended by the Angular team
  - Jasmine – "Jasmine is a behaviour-driven development framework for testing JavaScript code. It does not depend on any other JavaScript frameworks. It does not require a DOM. And it has a clean, obvious syntax so that you can easily write tests" [1]
  - Karma – A test runner written by the AngularJS team, Karma is well suited for integrating tests into the development process
  - Angular testing utilities – A number of tools used to create a controllable environment in which we can reliably test out code as it interacts with Angular
- Using the CLI sets up the test environment for you already, setting up any other way requires some configuration work on your part

Testing in Angular

- Pipes and Services can be tested in isolation
- Components can be tested in isolation but that will not expose how the component is interacting with Angular – including how it interacts with its own template and other components

Testing a Component

- Every test will be written in a *.spec.ts file as required by our karma.conf.js – this is a common convention and is born from the fact that in Jasmine tests are called Specs.
- Within the spec the first thing we do is import all the symbols we'll need for testing

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By }                        from '@angular/platform-browser';
import { DebugElement }              from '@angular/core';
Import { SomeComponent }             from `./some.component';
```

Testing a Component

- We then use Jasmine's describe method to contain the test suite

```
describe("A suite", function() {
   it("contains spec with an expectation", function() {
      expect(true).toBe(true);
   });
});
```

- We use Jasmine's beforeEach method to perform the setup before each spec in the describe block

```
beforeEach(function() {
   console.log('Running another spec')
});
```

Testbed

- Angular provides the TestBed which provides a testing module that we control and configure for this test through the use of the configureTestingModule function which receives an object not dissimilar to the @NgModule() decorator

```
beforeEach(() => {
   TestBed.configureTestingModule({
      declarations: [ SomeComponent ]
   });
});
```

- We then use the TestBed to create the component to be tested

```
fixture = TestBed.createComponent(SomeComponent);
component = fixture.componentInstance
```

Testbed

- The ComponentFixture provides us with access to the component instance itself and the DOM element of the component, the DebugElement

```
de = fixture.debugElement
```

- We can test the template in this way, particularly through use of the query function which takes a predicate function and returns the first element which satisfies it.

```
de = fixture.debugElement.query(By.css('h1'));
el = de.nativeElement;
```

- The 'By' object is an Angular testing utility that provides useful predicate functions like the one here

The specs (it)

- Jasmine's "it" function provides the specs and takes a string and function as arguments
- The forEach block runs before each spec

```
it('should exist',function(){
   expect(component).toEqual(jasmine.anything())
});
```

detectChanges

- During testing Angular does not automatically run change detection, we manually control it through calls to detectChanges()

```
it('should exist',function(){
   fixture.detectChanges();
});
```

- It is not until after the first call that data binding occurs and our template is populated with information from the model

Asynchronous testing

- In production you should be using external template and css files. These are loaded asynchronously and thus presents a problem for Jasmine as a synchronous testing suite.
- Angular fixes this by providing us with an async function we can pass to our beforeEach() blocks

```
beforeEach(async(() => {
   TestBed.configureTestingModule({
      declarations: [ SomeComponent ]
   }).compileComponents();
}));
```

- Don’t forget to import it – part of the core Angular testing module
- This abstracts us away from dealing with the asynchronous compilation of components and allows us to simply call compileComponents
- This asynchronous beforeEach should be placed ahead of the original synchronous beforeEach

Bringing it Together

```
import { TestBed, async } from '@angular/core/testing';
import { DebugElement } from '@angular/core';
import { By } from '@angular/platform-browser';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));

  it(`should have as title 'app works!'`, async(() => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app.title).toEqual('app works!');
  }));
});
```

Testing a Component with Dependencies

- Many components that we test will have dependencies on services that are provided. We will want to be able to test these components but in isolation of the real service.
- Testing the component without the real service makes testing much easier
- We will replace the real service with a mock or stub of some type and inject it using Angular

```
userServiceStub = {
        user: {
                firstname: “Chris”,
                surname: “Bruford”,
                company: “QA Ltd”
}
```

Testing a Component with Dependencies

- We need to register our stub with the injector. In our live application we do this within the decorator of a module or component, as we do here but using the configureTestingModule function

```
TestBed.configureTestingModule({
        …
        providers: [{provide: UserService, useValue: userServiceStub}]
});
```

- To inject the service we then need to use Angular's dependency injector. The safest way to do this is to use the injector of the component under test which can be found in the **DebugElement**

```
userService = de.injector.get(UserService)
```

- Alternatively if the service is registered with the root injector (as it is here) we can use the less verbose TestBed method

```
userService = TestBed.get(UserService)
```

Asynchronous calls

- Some of our services will make asynchronous calls which can be challenging for testing
- One solution often employed is to spy on the real service

```
spy = spyOn(userService, 'authenticate').and.callFake();
```

- We then need to write an asynchronous test which waits for asynchronous operations to complete through use of Angular's "whenStable" function

```
it('should update user details when logged in', async(()=>{
        userService.authenticate(…);

        fixture.whenStable().then(()=>{
            fixture.detectChanges();
            let content = el.querySelector('#userdetails > :first-child').textContent;
            expect(content).toContain('Test');
        })
    }));
```

Testing Inputs and Outputs

- Testing Inputs and Outputs requires us to confirm that the bindings work as expected and in this regard Angular provides a function to help us test Outputs, and testing Inputs requires us to simply change the input property and test the component responds as expected.
- We can test the component-under-test in isolation of its parent component, it's just not as useful as testing it with a parent component in play. So for a little extra work we can create a test parent component which gives us even great coverage.
    - Note: using an actual instance of the parent component is probably taking this too far and making too much work for too little gain

Testing Inputs and Outputs – Declaring the Parent

- So first we create the parent, or host, test component within our test suite

```
@Component({
	template: `<app-user-detail [user]="user" (favourite)="onFavourite(user)"></app-
user-detail>`
})
class TestHostComponent {
	favourites: User[] = [];
	user = {
		firstname: "Test",
		surname: "User",
		company: "Tests Unlimited Ltd"
	}

	onFavourite(user) {
		this.favourites.push(user);
	}
}
```

Testing Inputs and Outputs - Compiling

- We then set up the test suite similarly to before
- Note however that we create an instance of the parent component, which thereby implicitely creates an instance of the component-under-test as a child component within.

```
beforeEach(async(() => {
        TestBed.configureTestingModule({
                declarations: [UserDetailComponent, TestHostComponent]
        }).compileComponents();
}));

beforeEach(() => {
        fixture = TestBed.createComponent(TestHostComponent);
        component = fixture.componentInstance;
        fixture.detectChanges();
        userDetailDE = fixture.debugElement.query(By.css('.user-details'));
});
```

Testing Inputs and Outputs – Input Bindings

- Now we have access to both we can test inputs fairly simply by checking the property on the component-under-test has the data we bound in the parent element, and if used in the template we can check their too

```
it('should have the user details listed',()=>{
        let el = userDetailDE.nativeElement;
        let listItems = el.querySelectorAll('li');
        expect(listItems[0].textContent).toBe("Test");
        expect(listItems[1].textContent).toBe("User");
        expect(listItems[2].textContent).toBe("Tests Unlimited Ltd");
});
```

Testing Inputs and Outputs – Output Bindings

- Without user interaction we need to trigger the event that emits on the output property, fortunately Angular provides an easy solution for us to achieve this

```
it('should emit a favourite event when clicked', ()=>{
        expect(component.favourites.length).toBe(0);
        userDetailDE.triggerEventHandler('click', null);
        expect(component.favourites.length).toBe(1);
})
```

- Once we've triggered the even emission we simply look to the parent component to see if it carried through as expected

Testing HttpClient and Services – Set Up (1)

- Testing HTTP requests made by a service (and the HttpClient) require the HttpClientTestingModule
- The wrapping describe method requires a location service to be declared (and later initialised)
  - This needs to be provided in the TestBed configuration so that it can be used
- The HttpClientTestingModule must be provided in the TestBed's configuration

```
import { HttpClientTestingModule } from '@angular/common/http/testing';
…
describe('ServiceUnderTest', () => {
   let service: ServiceUnderTest;
   beforeEach(()=> {
      TestBed.configureTestingModule({
         providers: [ServiceUnderTest],
         imports: [HttpClientTestingModule]
      })
      …
   });
```

Testing HttpClient and Services – Set Up (2)

- Using a mocking strategy for data is common and this is done by importing Angular's HttpTestingController
- As with the service, this is declared at the start of the wrapping describe method and initialised in the beforeEach

```
import { HttpTestingController } from '@angular/common/http/testing';
…
describe('ServiceUnderTest', () => {
   let service: ServiceUnderTest;
   let httpMock: HttpTestingController;
   beforeEach(()=> {
      …
      service = TestBed.get(ServiceUnderTest);
      httpMock = TestBed.get(HttpTestingController);
   });
```

Testing HttpClient and Services – Service Methods

- To test a service's methods, a nested describe function is used with its own it function(s)
  - Usually provides some dummy data and makes a mock request to test the return of the method
  - Note: HTTP requests using HttpClient usually return an Observable of some class

```
…
describe('ServiceUnderTest', () => {
   …
   describe ('#getMethodUnderTest', () => {
      it('should return an Observable<MyClass>', () => {
         const dummyData: MyClass[] = [
            {dummy1: "Dummy1", dummy2: "Dummy2"},
            …
         ]
         …
      )}
   )};
)};
```

Testing HttpClient and Services – Service Methods

- The service's method is then subscribed to and we expect the result to equal the dummy data

```
…
describe('ServiceUnderTest', () => {
   …
   describe ('#getMethodUnderTest', () => {
      it('should return an Observable<MyClass>', () => {
         …
         service.getMethodUnderTest.subscribe(data => {
            expect(data).toEqual(dummyData);
         }
         …
      )}
   )};
)};
```

Testing HttpClient and Services – Service Methods

- To test that the correct request is made, we mock it by using the expectOne method and the URL the request is made to
- The request object can then be examined and tested (for example, for the correct method)
- The final part is to flush the created request of its data

```
…
describe('ServiceUnderTest', () => {
   …
   describe ('#getMethodUnderTest', () => {
      it('should return an Observable<MyClass>', () => {
         …
         const testRequest = httpMock.expectOne(`URL_to_test`);
         expect(testRequest.request.method).toBe('GET');
         testRequest.flush(dummyData);
      )}
   )};
)};
```

Exercise

WRITING TESTS BEFORE WRITING CODE