Introduction to Testing Angular Applications

Objectives

Key objectives of this chapter

- Angular testing technologies and setup
- Jasmine unit test basics
- Angular TestBed and test configuration
- Testing a service
- Testing a component

1.1 Unit Testing Angular Artifacts

- Just like any other development project, it is important to test Angular applications. This includes:
  - Manual unit testing during development
  - Automated unit testing. This will be the focus of this chapter.
  - Manual quality assurance testing by professional testers
- Angular applications are modular so with the right tools and techniques it is possible to perform robust unit/integration testing
- You are encouraged to write unit test scripts and run them after major changes to the codebase.
- Unit tests should also be run after a build to verify the build.

1.2 Testing Tools

- **Jasmine** - The Jasmine test framework is for testing JavaScript code
  - Jasmine is probably the most important tool used, besides Angular itself, as the unit tests themselves are written according to Jasmine
- **Karma** - Helps simplify running Jasmine tests although it can also work with other frameworks
  - Karma is a Node.js tool and requires Node.js and npm installed
  - You can run your Jasmine specifications (test scripts) in multiple browsers like Firefox and Chrome
  - Karma will automatically watch all JavaScript files and re-run the tests if any one of them is modified
- **Protractor** - Simulate user activities on a browser for "end to end" testing
  - This chapter will not focus on Protractor as there is not as much low-level integration required as with Jasmine
- **Angular testing utilities** - Angular provides API and tools to write unit tests.

Testing Tools

With Jasmine tests, you can generate an HTML "test runner" to run in a browser without Karma. Karma can automatically generate this for running tests and run the tests in a browser automatically, so Karma simplifies the testing process.

The main focus of this chapter is the Angular testing utilities and integration with Jasmine tests. Protractor testing is less Angular-specific and does not require the Angular testing utilities.

Jasmine – https://jasmine.github.io/

Karma – https://github.com/karma-runner/karma

Protractor – http://www.protractortest.org

1.3 Testing Setup

- Create a regular project to set it up for testing.

```
ng new simple-app
```

- That does these things:
  - It adds Karma and Jasmine dependencies in package.json.
  - Generates the Karma config file **karma.conf.js**.
  - It configures **.angular-cli.json** file so that a test script file is created by Angular CLI with every new component, service etc.
- A minimal project created using --minimal flag does not have support for testing. Do not use it for real projects.
- By default, Karma is configured to use the Chrome browser which must be installed.

Testing Setup

This chapter will focus on using the Angular project setup (or "quickstart") to setup for testing although Angular CLI setup should be similar.

The details of some of the configuration files above are not provided but can be found in the documentation of the relevant framework. This slide is mainly to be familiar with some of the files important in setting up testing.

There is also a 'protractor.config.js' file for Protractor configuration although that is not covered here.

Karma can integrate with other frameworks besides Jasmine but the Angular testing configuration uses this combination.

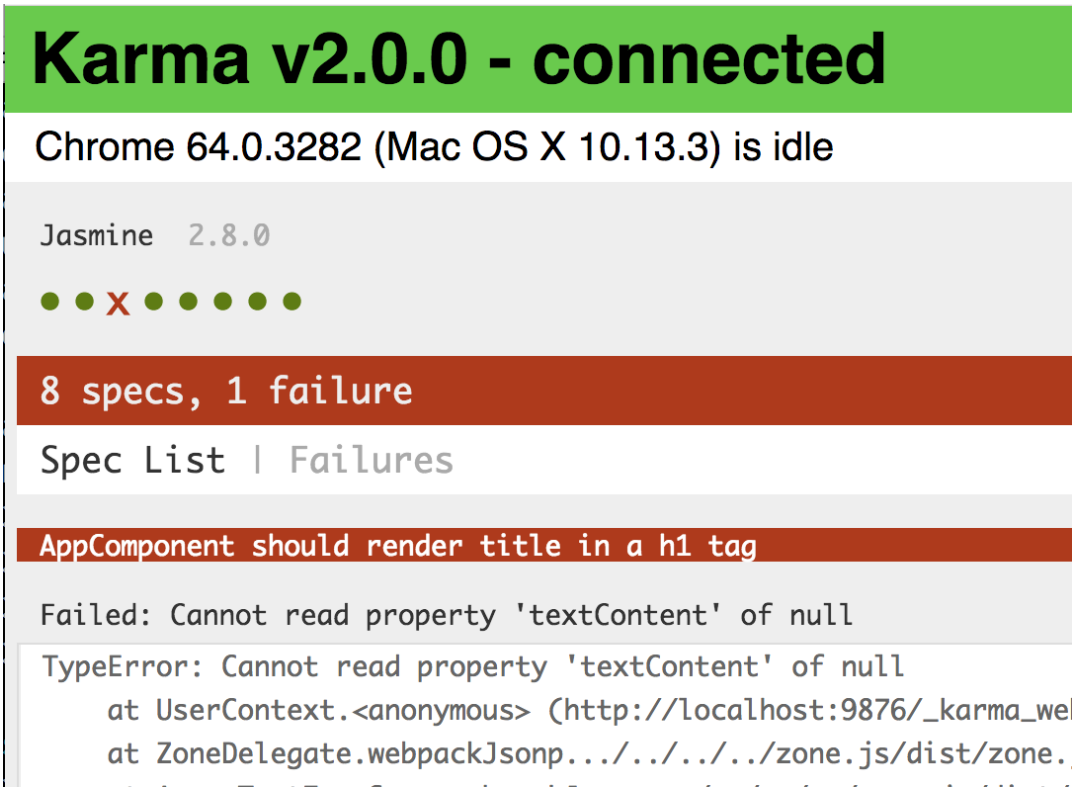
1.4 Typical Testing Steps

- Define test scripts. Angular CLI creates a file ending in '**.spec.ts**' for a generated component, service etc. Example:

```
banner.component.ts is tested by banner.component.spec.ts
```

- You can create additional test scripts. Just give them a '.spec.ts' extension.
- Run the '**npm test**' or '**ng test**' command to run the tests. This will:
  - Do a build.
  - Run all test scripts in the project.
  - Show the test result along with the application in a browser.
- Examine the test output for any test failures
- If you change your code the tests will be run again and the browser will be updated with the new test results
- End testing by hitting Control+C. This will also close the test browser.

1.5 Test Results



- Shows a green dot for each successful test and a red X for failed test.
- For each failed test the name of the test and a stack trace is shown

1.6 Jasmine Test Suites

- A test suite in Jasmine is defined with the **describe** function, which takes two parameters:
  - A string, which acts as a title (it is displayed on the spec runner page).
  - A function that implements the test suite:

```
describe("Test Suite #1", function() {
 // . . . unit tests (specs) or other describe functions
});
```

- A test suite acts as a container for:
  - Actual unit tests
  - Other test suites
    - So, you may have a hierarchy of nested test suites.
- Usually you write one test suite in each spec.ts file.

1.7 Jasmine Specs (Unit Tests)

- In Jasmine, a unit test is referred to as a **spec**.
- Specs are defined by the **it** function which takes two parameters:
  - A string, which acts as a title (it is displayed on the spec runner page facilitating the behavior-driven development).
  - A function that implements the actual unit test.
- Variables declared in the parent *describe* function (the test suite container) are visible in the unit tests (*it* function(s))
- A spec is a container for one or more expectations (assertions) about the code outcome (pass/fail).

```
describe('Simple test suite', () => {
    it("Test addition", () => {
        expect(1+2).toBe(3)
    })
})
```

1.8 Expectations (Assertions)

- Expectations are defined by the **expect** function that performs an assertion on the expected value of a variable or a function passed to the *expect* function as a parameter.
  - The *expect* function's parameter is called the **actual**.
- An expectation is evaluated to either *true* or *false.*
  - An expectation evaluated to *true* is treated as test success (test passed).
  - A *false* result is treated as a failed unit test.
- The *expect* function is chained with a *matcher* function, which takes the *expected* value.
- Example:

```
expect(<actual>).toBe(<expected value>);
```

where *toBe* is a *matcher* function chained to the *expect* function via the dot-notation.

- **Note**: You can also chain other matchers.

1.9 Matchers

- A **matcher** is a function that performs a boolean comparison between the actual value (passed to the chained *expect* function) and the expected value (passed as a parameter to the *matcher*).
- Jasmine comes with a rich catalog of built-in matchers:

```
toBe                        toBeUndefined
toBeCloseTo                 toContain
toBeDefined                 toEqual
toBeFalsy                   toHaveBeenCalled
toBeGreaterThan             toHaveBeenCalledWith
toBeLessThan                toMatch
toBeNaN                     toThrow
toBeNull                    toThrowError
toBeTruthy
```

- Developers can also create their own (custom) matchers.

Matchers

toBe compares using the triple equal sign: ===

toEqual works for simple literals and variables

toMatch is used for regular expressions

toBeDefined and **toBeUndefined** compare against *undefined*

toBeNull compares against null

toBeTruthy checks for true

toBeFalsy checks for false

toContain is used for finding an element in an Array

toBeLessThan is used for mathematical '<'

toBeGreaterThan is used for mathematical '>'

toBeCloseTo is used for precision math comparison

toThrow is used for testing if a function throws an exception

1.10 Examples of Using Matchers

- Here is a full example of a spec used to test a simple function

```
it ("Test #234", function(){

    var testFunction = function (){
        return 1000 + 1;
    };

    expect(testFunction()).toEqual(1001);
});
```

  - **Note**: The *testFunction* function would normally be placed in a separate JavaScript file of functions to be tested.
- Similarly, you can test a variable.
- **Note**: The *expect* function performs an expression evaluation, so you can assert math expressions as well, e.g.

```
expect(1000 + 1).toEqual(1001);
```

1.11 Using the not Property

- To reverse the expected value (e.g. from *false* to *true*), matchers are chained to the *expect* function via the **not** property.
- For example, the following assertions are functionally equivalent:

```
expect(<variable or a function>).toBe(true);
expect(<variable or a function>).not.toBe(false);
```

1.12 Setup and Teardown in Unit Test Suites

- Jasmine lets you write functions that are called before and after each spec (test). They are registered using *beforeEach()* and *afterEach()*.
- The *beforeEach* function runs the common initialization code (if needed).
- The *afterEach* function contains the clean up code (if needed).
- Both functions take a function as a parameter.
- You can have multiple beforeEach() and afterEach() functions.

1.13 Example of beforeEach and afterEach Functions

```
describe("A Test suite with setup and teardown", function(){
  var obj = {};

  beforeEach(function() {
    obj.prop1 = true;
    // obj.fooBar - undefined !
  });

  afterEach(function() {
    obj = {};
  });

  it("has one property, for sure", function() {
    expect(obj.prop1).toBeTruthy(); // is true, indeed
  });

  it("has undefined property", function() {
    expect(obj.fooBar).toBeUndefined();
  });
});
```

Example of beforeEach and afterEach Functions

None of the code above is specific to Angular. This just shows the Jasmine-related code and the structure of a Jasmine test.

1.14 Angular Test Module

- Every test suite needs to define an Angular module.
- Tests are run using this module and not the real application module. This is necessary so that you can use mock services during testing.
- The item under test and all of its dependencies must be added as providers, declarations, and imports of that module.
- A test module is created using the **configureTestingModule()** method of the **TestBed** class.

1.15 Example Angular Test Module

```
import { TestBed, inject } from '@angular/core/testing';
import { HttpClientModule } from '@angular/common/http';
import { BlogService } from './blog.service';

describe('BlogService', () => {
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientModule],
      providers: [BlogService]
    });
  });
//...
}
```

- Here we are testing BlogService which depends on HttpClient. Our test module had to be set up for that to work.

1.16 Testing a Service

- Services are easier to test than components. This should encourage you to write more test scripts for services.
- We will test this service.

```
export class BlogService {
  constructor(private http:HttpClient) { }

  sayHello(name:string) : string {
    return `Hello ${name}`
  }

  getBlogPost(postId:number) : Observable<Object> {
    return this.http.get<Object>(`/posts/${postId}`)
  }
}
```

1.17 Injecting a Service Instance

- Before we can call service methods we need to obtain an instance through injection.
  - We can't simply create an instance using new. Doing so will not perform injection within the service if it depends on other services such as HttpClient.
- Among the many ways to inject a service the easiest is to call **TestBed.get()**.

```
describe('BlogService', () => {
  let service:BlogService

  beforeEach(() => {
    TestBed.configureTestingModule({...});

    service = TestBed.get(BlogService)
  })

  it("Should use injection", () => {
    expect(service).toBeTruthy()
  })
}
```

1.18 Test a Synchronous Method

- Nothing special about testing a synchronous method. Just call the method and verify its behavior.

```
it("Should call sayHello", () => {
  let name = "Bob"
  let greeting = service.sayHello(name)

  expect(greeting).toBe(`Hello ${name}`)
})
```

1.19 Test an Asynchronous Method

- There are several ways to test an asynchronous method in Angular. The easiest is to use the done() callback. It is a callback that is passed to an it() arrow function. We need to call the done function to indicate when the asynchronous test has completed.

```
it("Should get blog post", (done) => {
  service.getBlogPost(1).subscribe((blogPost) => {
    expect(blogPost["id"]).toBe(1)
    expect(blogPost["title"]).toBe("My cool blog post")

    done()
  })
})
```

1.20 Using Mock HTTP Client

- To ensure a consistent response from a web service during testing you may need to supply canned responses right from the test script.
- Angular gives us the **HttpClientTestingModule** that provides an alternate implementation of the HttpClient service. This mock implementation lets us supply static local data as responses to HTTP calls.
- From the test module import HttpClientTestingModule instead of HttpClientModule.

```
import {
  HttpClientTestingModule,
  HttpTestingController
} from '@angular/common/http/testing'

beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    providers: [BlogService]
  });
});
```

1.21 Supplying Canned Response

```
it("Should get blog post", (done) => {
  service.getBlogPost(1).subscribe((blogPost) => {
    expect(blogPost["title"]).toBe("Mock title")
    expect(blogPost["body"]).toBe("Mock body")

    done()
  })

  let controller: HttpTestingController =
    TestBed.get(HttpTestingController)

  //Get a mock request for the URL
  let mockRequest = controller.expectOne("/posts/1")

  //Supply mock data
  mockRequest.flush({
    "id": 1,
    "title": "Mock title",
    "body": "Mock body"
  })
})
```

Notes

The expectOne() method returns a TestRequest object for a given URL. This mock request can be used to supply canned response data.

Finally, the TestRequest.flush() method is used to supply mock response data.

Note: With the alternate implementation of the HttpClient service you must provide local data as response. If you don't the subscriber will never receive any response.

1.22 Testing a Component

- Here we will test a very simple component.

```
@Component({
  selector: 'app-greet',
  templateUrl: './greet.component.html'
})
export class GreetComponent {
  customerName = "Daffy Duck"

  welcomeBugsBunny() {
    this.customerName = "Bugs Bunny"
  }
}

//Template
<p>Welcome {{customerName}}</p>
<button (click)="welcomeBugsBunny()">Bugs is Here!</button>
```

1.23 Component Test Module

- Set up a test module and add all services and modules the component depends on.
- Compile the template of the component. External templates are compiled asynchronously. So we need to wrap the who module definition in an **async()** call.

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    imports: [FormsModule, ...],
    providers: [...]
    declarations: [ GreetComponent ]
  })
  .compileComponents();
}));
```

1.24 Creating a Component Instance

- A component is tested in isolation and not as a part of its parent component's template. We need to create an instance of the component using **TestBed.createComponent()**. This gives us a **ComponentFixture** object which gives us access to the actual component instance and much more.
- Angular CLI generates the necessary code. Here it is for review.

```
describe('GreetComponent', () => {
  let component: GreetComponent;
  let fixture: ComponentFixture<GreetComponent>;
//...
  beforeEach(() => {
    fixture = TestBed.createComponent(GreetComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });
}
```

1.25 The ComponentFixture Class

- Commonly used properties:
  - **componentInstance** - The component instance.
  - **nativeElement** - The DOM Element object that is at the root of the DOM generated by the component. We can use the DOM API to verify its correctness.
  - **debugElement: DebugElement** - Provides utility methods useful for testing.
- Useful methods:
  - **detectChanges()** - By default Angular does not do component state change detection during testing. We need to call this to initiate change detection.
  - **autoDetectChanges(true)** - Enable automatic change detection.

1.26 Basic Component Tests

- Verify initial state.

```
it('Initial state', () => {
  expect(component.customerName).toBe("Daffy Duck");
});
```

- Verify component state change.

```
it('State change', () => {
  component.customerName = "Bob" //Change state

  fixture.detectChanges(); //Trigger state change handling

  let paraText =
    fixture.nativeElement.querySelector('p').textContent
  expect(paraText).toBe('Welcome Bob');
});
```

Notes

Here **querySelector('p').textContent** is a standard DOM API used to get the contents of the <p> tag.

1.27 The DebugElement Class

- This class provides utilities to help write test scripts.
- Useful properties:
  - **componentInstance** - Same as ComponentFixture.componentInstance
  - **nativeElement** - Same as ComponentFixture.nativeElement
  - **parent : DebugElement** - The parent element.
- Useful methods:
  - **query()/queryAll()** - Obtain child DebugElement by CSS selector query.
  - **triggerEventHandler()** - Used to simulate user interactions like button clicks.

1.28 Simulating User Interaction

- Here we trigger a button click event and verify the change in the component's state.

```
import { By } from '@angular/platform-browser';

it('Should handle click', () => {
  let button : DebugElement =
    fixture.debugElement.query(By.css("button"))

  button.triggerEventHandler("click", null)

  fixture.detectChanges();

  expect(component.customerName).toBe("Bugs Bunny")

  let paraText =
    fixture.nativeElement.querySelector('p').textContent
  expect(paraText).toBe('Welcome Bugs Bunny');
});
```

1.29 Summary

- Jasmine and Karma are the two main technologies used in Angular component unit testing
  - Test code is written using Jasmine
  - Karma provides automation for configuring and running the tests
- Jasmine tests are defined within 'describe' functions that have 'beforeEach' functions for test setup and 'it' functions for the actual tests
- The Angular TestBed class is the most important class for test configuration and execution
- A separate Angular module is created for each test suite. Tests are run using that module and not using the application module.
- Angular provides us ways to inject service instances and test synchronous and asynchronous methods.
- Components are tested in isolation and not as a part of a larger application. We can use standard DOM API to test the validity of the DOM generated by a component.