Unit Testing Angular Like a Boss

Workshop Overview

Workshop Schedule

- Intro
- Karma Test Runner
- Isolated Unit Tests
- Integration Unit Tests
- DOM Interaction
- Routing Components
- Advanced Topics

Why Are You Here?

Workshop Goal

- Learn Angular Unit Testing - Isolated & Integration
- Learn how to write good tests
- Make You Awesome

Don’ts

- Violate the CoC
- Type along while I type

DO’s

- Watch & Learn
- Take Notes
- Work in Pairs/Groups

Introduction to Testing

What is Automated Testing

- Unit Tests
- Integration/Functional Tests
- End to End Tests

Benefits of Testing

- Documented Intentions
- Improved Design
- Fewer Bugs into Production
- No Regressions
- Safer Refactoring

Unit Testing

Testing Tools

- Frameworks
  - Jasmine
  - Mocha
- Assertion Libraries
  - Jasmine
  - Chai
  - should.js
  - Expect
- Mocking Libraries
  - sinon.js
  - Testdouble.js
  - Jasmine
- Test Runners
  - Karma
  - Jest
  - Cypress

Jasmine

- Behaviour Driven Development (BDD) Testing Framework
- Describe your code via specs
  - `describe(...)`
  - `it(...)`
  - `beforeEach(...)`
- Assert via expectations
  - `expect(...)...`
- Using matchers
  - `toEqual(...)`
  - `not.toEqual(...)`
  - `toThrow(...)`
  - `toContain(...)`
- `Exclusions/Forcing`
  - `xdescribe, xit`
  - `fdescribe, fit`

```
describe("join function", function() {
  var joiner;

  beforeEach(function() { joiner = new Joiner(); });

  it("should join an array with a separator", function() {
    var joined = joiner.join([1,2], '-');
    expect(joined).toEqual('1-2');
  });

  it("should default to a comma string separator", function() {
    var joined = joiner.join([3,4]);
    expect(joined).toEqual('3,4');
  });

  it('should error when not passed an array', function() {
    expect(function() { joiner.join({}, ',') }).toThrow();
  });
});
```

Writing Good Tests

How to structure tests

- **Arrange** all necessary preconditions and inputs.
- **Act** on the object or method under test.
- **Assert** that the expected results have occurred.

DAMP and DRY

- **DRY - Don't Repeat Yourself**
  - Promotes the removal of duplication in the code
  - Isolate change to those parts of the system that must change
- **DAMP**
  - Some minor amount of duplication
- **RULES**
  - Repeat yourself if necessary to make it easier to read
  - Minimize logic out of tests (what will test the tests?)

Tell the Story

- A test should be a complete story, all within the `it()`
- You shouldn’t need to look around much to understand the test
- Techniques
  - Remove less interesting setup to `beforeEach()`
  - Keep critical setup within the `it()`
  - Include all of the "Act" and "Assert" test parts are in the `it()` clause

Overly DRY Test

- `ngOnInit` is crucial to the story of the test
- But it is hidden away in the `beforeEach` block
- The tests don't have any "Action"

```
describe("Hero Detail Component", function() {
  var heroDetCmp;

  beforeEach(function() {
    heroDetCmp = createComponent();
    heroDetCmp.ngOnInit();
  });

  describe('ngOninit' function() {

    it("should set the hero", function() {
      expect(heroDetCmp.hero).toBeDefined()
    });

    it("should set the heroId", function() {
      expect(heroDetCmp.heroId).toBe(3));
    });
  });
});
```

DAMP Test

- Creating the component is just setup
- Calling `ngOnInit` is important to the story of the test

```
describe("Hero Detail Component", function() {
  var heroDetCmp;

  beforeEach(function() {
    heroDetCmp = createComponent();
  });

  describe('ngOninit' function() {

    it("should set the hero", function() {
      heroDetCmp.ngOnInit();

      expect(heroDetCmp.hero).toBeDefined()
    });

    it("should set the heroId", function() {
      heroDetCmp.ngOnInit();

      expect(heroDetCmp.heroId).toBe(3));
    });
  });
});
```

Unit Testing Best Practices

Isolating the Unit in tests

- Only test the unit and not its dependents or dependencies
- Use **Test Doubles** to isolate dependencies
  - **Mocks**: objects pre-configured with details of the calls they expect
  - **Spies**: record information about calls
  - **Stubs**: provide canned answers to calls made during the test
  - **Dummies**: objects that are passed around but never actually used.

Jasmine Spies

- Jasmine Spies can provide **stub** and **spy** behaviour

```
var getUser = jasmine.createSpy('getUser');

getUser.and.returnValue({ name: 'Sam'});
var user = getUser(123);
expect(user).toEqual({name: 'Sam'});
expect(getUser).toHaveBeenCalledWith(123);
```

Creating Spies

```
class BankAccount {
    getBalance() { … }
    deposit() { … }
    withdraw() { … }
}

jasmine.createSpyObj(['getBalance', 'deposit', 'withdraw']);
```

Unit Testing in Angular

How deep to test?

- **Isolated tests**: only the class, mocking everything
- **Integration tests:** compiling components and using the injector
  - **Shallow**: mock out related components
  - **Deep**: include all components

Isolated Unit Tests

Isolated Tests

- Just like plain JavaScript testing
  - No special tools
  - Use mocks to isolate the unit
  - Name tests with “.spec.ts”

Interaction Testing

Interaction Testing

- State based testing
  - Change state
  - Assert that some piece of state has changed
- Interaction based testing
  - Call a method on another class
  - Assert that a method was called correctly

Interaction Testing of Code

```
expect(class.methodA).toHaveBeenCalled();

expect(class.methodB).toHaveBeenCalledWith(value);
```

Integration Unit Tests

NgModule

- Templates are compiled by the Angular Compiler
  - This can be done dynamically at runtime
  - Or in a build step by the Ahead of Time (AOT) Compiler - via `ngc` tool
- The unit of compilation is called an `NgModule`, which specifies what:
  - Template stuff (components, directives and pipes) to compile
  - Other `NgModules` to be imported
  - Template stuff and modules to export for other `NgModules` to import
  - Components to be bootstrapped
  - Services to provide to the injector
- An Angular App is defined by a root `NgModule`

Angular Testing Utilities

- `TestBed` - a harness for compiling components
- `inject()` - provides access to injectables
- `async()` & `fakeAsync()` - async Zone control

TestBed - configure Testing Module

- TestBed configures a temporary NgModule for testing

```
TestBed.configureTestingModule({
  declarations: [ HeroComponent ],
  imports: [ ... ],
  providers: [ ... ],
  Schemas: [ ... ]
});
```

TestBed - creating a component

- TestBed creates the component in a ComponentFixture

```
fixture = TestBed.createComponent(HeroComponent);
```

Component Fixture

- Access to the component, its DOM and change detection
  - `componentInstance` - the instance of the component created by TestBed
  - `debugElement` - provides insight into the component and its DOM element
  - `nativeElement` - the native DOM element at the root of the component
  - `detectChanges()` - trigger a change detection cycle for the component
  - `whenStable()` - returns a promise that resolves when the fixture is stable

Debug Element

- Insights into the component's DOM representation
  - **`parent` / `children`** - the immediate parent or children of this `DebugElement`
  - **`query(predicate)`** - search for one descendant that matches
  - **`queryAll(predicate)`** - search for many descendants that match
  - **`injector`** - this component's injector
  - **`listeners`** - this callback handlers for this component's events and `@Outputs`
  - **`triggerEventHandler(listener)`** - trigger an event or `@Output`

inject(tokens, fn)

- Gets services from the root injector
- Can be placed in **`beforeEach`** or **`it`** blocks:

```
let heroService;
beforeEach(inject([HeroService], (service: HeroService) => {
  heroService = service;
}));
```

- To get services from the component injector use:

```
const service = fixture.debugElement.injector.get(token);
```

TestBed.get(Type)

- Gets services from the root injector
- Can be placed in **`beforeEach`** or **`it`** blocks:

```
let heroService;
beforeEach(() => {
  heroService = TestBed.get(HeroService);
}));
```

Shallow Integration Tests

- Build the component via the TestBed
- Mock out or ignore other components

Querying the DOM

- NativeElement provides:
  - `querySelector(cssSelector)`
- DebugElement provides:
  - `query(predicate)`
  - `queryAll(predicate)`
- Predicates can be created by helpers:
  - `By.css(selector)`
  - `By.directive(DirectiveType)`

Interacting with the DOM

- `nativeElement` - can't use outside the browser
  - `dispatchEvent`
  - `textContent`
- `debugElement` - doesn't have access to `textContent`
  - `triggerEventHandler`
  - `properties`
  - `attributes`
  - `classes`
  - `styles`

Deep Integration Tests

Deep Component Testing

- Nested Components need to be tested too
- Shallow testing (mocking all children) is not enough
- Deep tests check that
  - the parent is rendering the children correctly
  - the child is receiving the correct values in its inputs
  - the parent handles output events correctly

Access to the child components

- Search for instances of the child component:

```
const heroElements = fixture.debugElement.queryAll(By.directive(HeroComponent));
```

- Check the value of `@Input` properties on the child component :

```
expect(heroComponents[0].componentInstance.hero).toBe(HEROES[0]);
```

- Trigger `@Output` bindings :

```
heroComponents[0].triggerEventHandler('delete', null);
```

Integration Testing of Services

- Use the `TestBed` to configure dependencies
- Mock the `Http` service
  - `HttpClientTestingModule`
  - `HttpTestingController`

Asynchronicity in Unit Tests

Observables vs Promises

- A promise represents: A **single** value in the **future**.
- An observable represents: **Zero or more** values **now or in the future**.
- THEREFORE
  - Promises must be asynchronous (like setTimeout, setInterval, user interactions)
  - Observables can be asynchronous or synchronous

Jasmine and Async tests

- Adding a **done** parameter to an **it** clause makes a spec async

```
it('should do something async', done => {
  let value;
  setTimeout(() => value = 42, 100);
  setTimeout(() => {
    expect(value).toBe(42);
    done();
  }, 200);
  expect(value).toBeUndefined();
});
```

Zone.js

- Intercepts and tracks asynchronous callbacks
  - Intercept asynchronous task scheduling
  - Wrap callbacks for error-handling and zone tracking across async operations.
  - Provide a way to attach data to zones
  - Provide a context specific last frame error handling
- Configured by rules (or specs)
  - `AsyncTestZoneSpec` - rules for async test zones
  - `FakeAsyncTestZoneSpec` - rules for fake async test zones

async(...) helper

- Wraps a test function in an asynchronous test zone.
- Test automatically completes when all async calls in this zone are done.
- *Not great for unit tests if the async delays are long.*

```
it('should do something async', async(() => {
  let value;
  setTimeout(() => value = 42, 100);
  setTimeout(() => expect(value).toBe(42), 200);
  expect(value).toBeUndefined();
}));
```

fakeAsync(...) helper

- Wraps a test function in a fake asynchronous test zone.
- All async calls are captured in a list that can be flushed synchronously.
- *Great for fine-grain control over asynchronicity.*

```
it('should do something async', fakeAsync(() =>
{
  let value;
  setTimeout(() => value = 42, 100);
  setTimeout(() => value = 84, 200);
  tick(100);
  expect(value).toBe(42)
  tick(100);
  expect(value).toBe(84)
}));
```