SECTION 03 - ANGULAR COMPONENT TESTING IN DEPTH

Index

- Intro

- We're going to start with the simplest form of angular components, which are representational components.

- We will start with **synchronous component tests** and then we are going to move on to **asynchronous component testing**.

- **Presentational components** simply take some input data and then displayed on the screen. Due to the simplicity of this type of components, many projects opt by not testing presentational components and simply focusing the test effort on more complex components. This is a valid approach. However, in this lesson we are going to show how to test these type of components anyway. We are going to do this for two reasons:
  - First, you might have to test them due to test coverage requirements in your project that are not under your control.

  - Also we are going to use this simple type of component just to show how an angular unit test can validate the content of the DOM if needed.

- In CoursesModule, imported in app.module.ts, all the components that our application needs are imported.

- TestBed to configure component tests

- TestBed is used to configure the component tests.

- In order for this to work, it's important that the courses.module to import the **CommonModule**, which contains directives such as *ngIf, *ngFor, etc. that are used almost in every component.

- Unlike the app.module.ts, in the courses.module there isn't anything that would prevent our tests from running, such as, for example, BrowserModule or the HttpClientModule.

- Taking the CoursesModule we have everything that our component needs to execute in this testing environment.

```
@NgModule({
    declarations: [
      AppComponent
    ],
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        HttpClientModule,
        MatToolbarModule,
        MatButtonModule,
        CoursesModule,
        AppRoutingModule
    ],
    providers: [
    ],
    bootstrap: [AppComponent]

})
export class AppModule {
}
```

```
TS courses.module.ts ×
src > app > courses > TS courses.module.ts > CoursesModule

import { NgModule } from '@angular/core';…

@NgModule({
  declarations: [
    HomeComponent,
    AboutComponent,
    CourseComponent,
    CoursesCardListComponent,
    CourseDialogComponent
  ],
  imports: [
    CommonModule,
    MatMenuModule,
    MatButtonModule,
    MatToolbarModule,
    ■ ■ ■
    MatMomentDateModule,
    ReactiveFormsModule,
    AppRoutingModule
  ],
  exports: [
    HomeComponent,
    AboutComponent,
    CourseComponent,
    CoursesCardListComponent,
    CourseDialogComponent
  ],
  providers: [
    CoursesService,
    CourseResolver
  ],
  entryComponents: [CourseDialogComponent]

})export class CoursesModule {
```

```
TS courses-card-list.component.spec.ts U ×
src > app > courses > courses-card-list > TS courses-card-list.component.spec.ts >
 1 > import { async, ComponentFixture, TestBed } f
10
11  describe("CoursesCardListComponent", () => {
12    beforeEach(() => {
13      TestBed.configureTestingModule({
14        imports: [CoursesModule],
15      });
16    });
```

- Create an instance of a component for each test

- The compilation of angular components is an asynchronous process that in certain cases might even trigger external HTTP requests to fetch e-mail templates, stylesheets, etc. So the only safe way to execute some code after the compilation has finished is to use "*then*" block and wait for the promise to resolve.

- In order to create an instance of a component, we are going to need a **Component Fixture**. So the Component Fixture is a **test utility type** that is going to help us to

do some common test operations, such as, for example, obtaining an instance of the component, debugging the component, etc.

```
let component: CoursesCardListComponent;
let fixture: ComponentFixture<CoursesCardListComponent>;

beforeEach(() => {
 TestBed.configureTestingModule({
   imports: [CoursesModule],
 })
   .compileComponents()
   .then(() => {
     fixture = TestBed.createComponent(CoursesCardListComponent);
     component = fixture.componentInstance;
   });
});
```

- Solving asynchronous problems with waitForAsync()

- This is a ***synchronous before-each block***, meaning that the promise (compileComponents()) is going to get evaluated and the two variables (*fixture* and *component*) are going to be filled in.

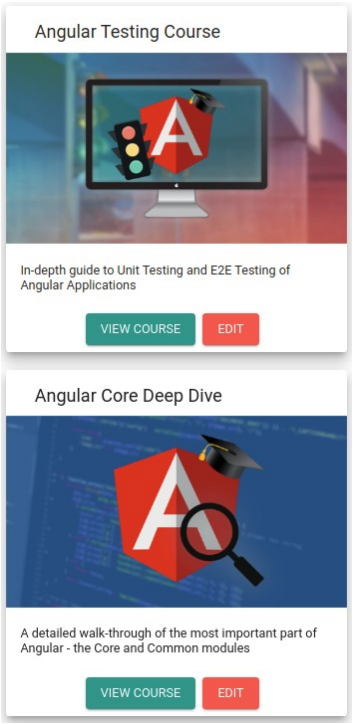
- The problem is that the test runner is not going to wait for the promise to get resolved before proceeding with the execution of the test specifications of the test suite.

- So what's happening is that the body inside the then() method that only gets executed whenever the promise gets evaluated, is only to be executed after the execution of the test itself and not before like we intended when we put this code inside *before-each*. The solution to this issue is using **waitForAsync**.

```
beforeEach(
 waitForAsync(() => {
   TestBed.configureTestingModule({
     imports: [CoursesModule],
   })
     .compileComponents()
     .then(() => {
       fixture = TestBed.createComponent(CoursesCardListComponent);
       component = fixture.componentInstance;
     });
 })
);

it("should create the component", () => {
 expect(component).toBeTruthy();
 console.log("something here");
});
```

- Query the DOM

- To obtain the list of *divs* that contains each of the course cards:



- As we can see from the template, each card has a "*course-card*" class:

```
TS courses-card-list.component.spec.ts ●   <> courses-card-list.component.html ×   TS setup-test-data.ts   TS db-d
> courses > courses-card-list > <> courses-card-list.component.html > mat-card.course-card.mat-elevation-z10

<mat-card *ngFor="let course of courses" class="course-card mat-elevation-z10">
```

- Setting up the test environment for querying the DOM:

```
describe("CoursesCardListComponent", () => {
  let component: CoursesCardListComponent;
  let fixture: ComponentFixture<CoursesCardListComponent>;
  let el: DebugElement;

  beforeEach(
    waitForAsync(() => {
      TestBed.configureTestingModule({
        imports: [CoursesModule],
      })
        .compileComponents()
        .then(() => {
          fixture = TestBed.createComponent(CoursesCardListComponent);
          component = fixture.componentInstance;
          el = fixture.debugElement;
        });
    })
  );
```

- After assigning any data to a component via an input property, we also need to notify the component that some changes were made.

```
it("should display the course list", () => {
  component.courses = setupCourses();   Assign an array into the input property of the component

  fixture.detectChanges();

  console.log(">>> " + el.nativeElement.outerHTML);   To inspect the component

  const cards = el.queryAll(By.css(".course-card"));   Query the DOM

  expect(cards).toBeTruthy("Could not find cards");
  expect(cards.length).toBe(12, "Unexpected number of courses");
});
```

- Debug a component

- From inside a spec: console.log(">>> " + el.nativeElement.outerHTML);
- Then, by inspecting the browser in which Karma is running:

```
>>> <div id="root1" _nghost-a-c224="" ng-version="12.0.1"><!- context.js:265
-container--></div>
```

→ In this case I can see how the element is empty.

- Container Components vs Presentational Components

- A Smart or Container Component does not have any presentational responsibilities.
- A Container Component is typically a top level component that fetches the data that the application needs from a service such as, for example, the CoursesService that we have tested before. So an example of this type of components is the HomeComponent that displays, among other things, the CoursesCardListComponent.
- The Container Component receives the data from a service and not from an input. This is very different than a Presentational Component such as the CoursesCardListComponent that gets its data only from inputs.
- The Presentational Component could be easily reused in multiple parts of the application whenever we need to display at least, of course, cards. On the other hand, the HomeComponent is really specific here to the home page.
- Notice that even though in this course we talk about Presentation Components versus Container Components, in reality, not every component needs to fall strictly into one of the two categories. Talking about Presentational Components versus Container Components is just a useful way of discussing what is the main responsibility of each component.

- Testing Container Components

- Given the suit:

→ Previous setup:

```
describe("HomeComponent", () => {
  let fixture: ComponentFixture<HomeComponent>;
  let component: HomeComponent;
  let el: DebugElement;
  let coursesService: any;

  const beginnerCourses = setupCourses().filter(
    (course) => course.category == "BEGINNER"
  );

  beforeEach(
    waitForAsync(() => {
      const coursesServiceSpy = jasmine.createSpyObj("CoursesService", [
        "findAllCourses",
      ]);
      TestBed.configureTestingModule({
        imports: [CoursesModule, NoopAnimationsModule],
        providers: [{ provide: CoursesService, useValue: coursesServiceSpy }],
      })
        .compileComponents()
        .then(() => {
          fixture = TestBed.createComponent(HomeComponent);
          component = fixture.componentInstance;
          el = fixture.debugElement;
          coursesService = TestBed.inject<CoursesService>(CoursesService);
        });
    })
  );
```

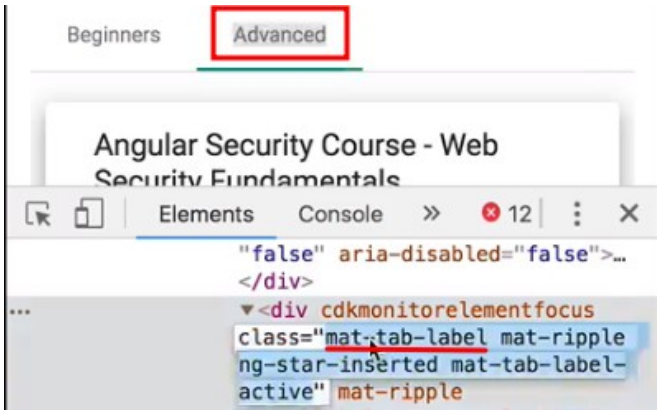
```
it("should display only beginner courses", () => {
 coursesService.findAllCourses.and.returnValue(of(beginnerCourses));
 fixture.detectChanges();

 const tabs = el.queryAll(By.css(".mat-tab-label"));
 expect(tabs.length).toBe(1, "Unexpected number of tabs found");
});
```

→ Instead of returning the array and pass it inside the *returnValue()* method, let's return an Observable that immediately *emits* the value. For that, we use the Array JS "***of***" factory method. This is going to create an Observable that takes the array, *emits* it immediately and then it completes. And all of this is going to happen synchronously and not asynchronously.

```
export class CoursesService {

    constructor(private http:HttpClient) {

    }

    findCourseById(courseId: number): Observable<Course> {
        return this.http.get<Course>(`/api/courses/${courseId}`);
    }

    findAllCourses(): Observable<Course[]> {
        return this.http.get('/api/courses')
            .pipe(
                map(res => res['payload'])
            );
    }
```

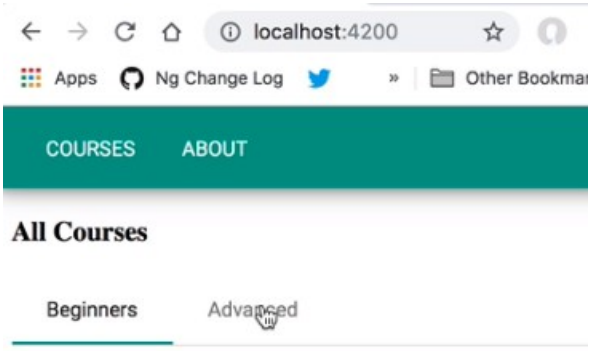
→ This is why we need to use *of()* method

```
spec.ts  home.component.html  db-data.ts  home.component.ts  home.compon
s > home > home.component.html > div.container > mat-tab-group

<ng-container *ngIf="(beginnerCourses$ | async) as beginnerCourses">

  <mat-tab label="Beginners" *ngIf="beginnerCourses?.length > 0">

    <courses-card-list (courseEdited)="reloadCourses()"
                       [courses]="beginnerCourses">

    </courses-card-list>

  </mat-tab>

</ng-container>


<ng-container *ngIf="(advancedCourses$ | async) as advancedCourses">

  <mat-tab label="Advanced" *ngIf="advancedCourses?.length > 0">

    <courses-card-list (courseEdited)="reloadCourses()"
                       [courses]="advancedCourses">


    </courses-card-list>

  </mat-tab>

</ng-container>
```



- Simulating User DOM Interaction in Angular Unit Tests

- The interaction that we are simulating is to have a user interact with the screen with both beginner and advanced courses. And if the user clicks on the advanced, then we are going to confirm via a test that indeed the advanced courses are now getting displayed.



→ STILL THIS TEST IS NOT WORKING

```
it("should display advanced courses when tab clicked", () => {
 coursesService.findAllCourses.and.returnValue(of(setupCourses()));
 fixture.detectChanges();

 const tabs = el.queryAll(By.css(".mat-tab-label"));
 el.nativeElement.click(tabs[1]);
 // click(tabs[1]); another option using the custom utility function
 const cardTitles = el.queryAll(By.css(".mat-card-title"));

 fixture.detectChanges();

 expect(cardTitles.length).toBeGreaterThan(0, "Could not find card titles");
 expect(cardTitles[0].nativeElement.textContent).toContain(
   "Angular Security Course"
 );
});
```

→ In the next section, we are going to understand exactly what is the problem with this failing test and we're going to fix it. That is going to be the prelude for a complete section on angular synchronous testing.