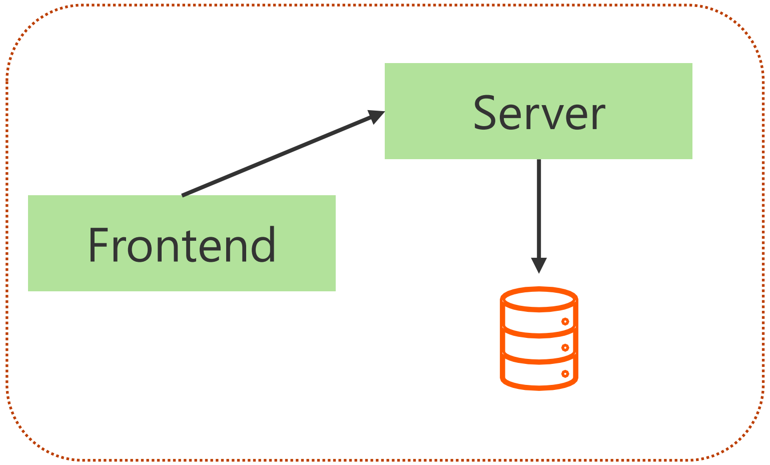
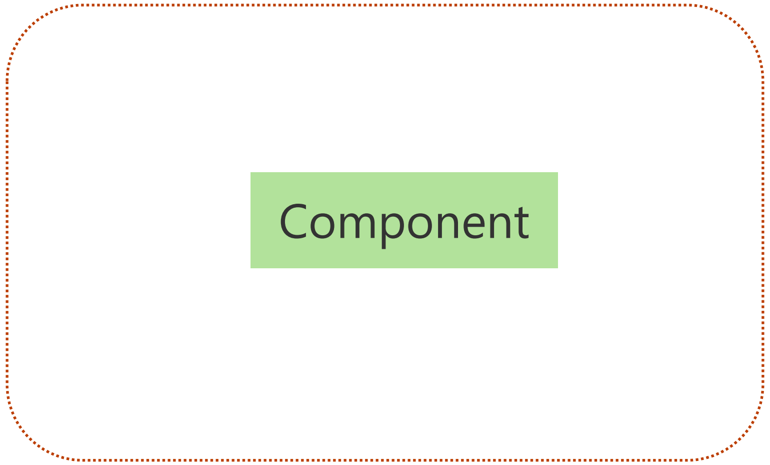
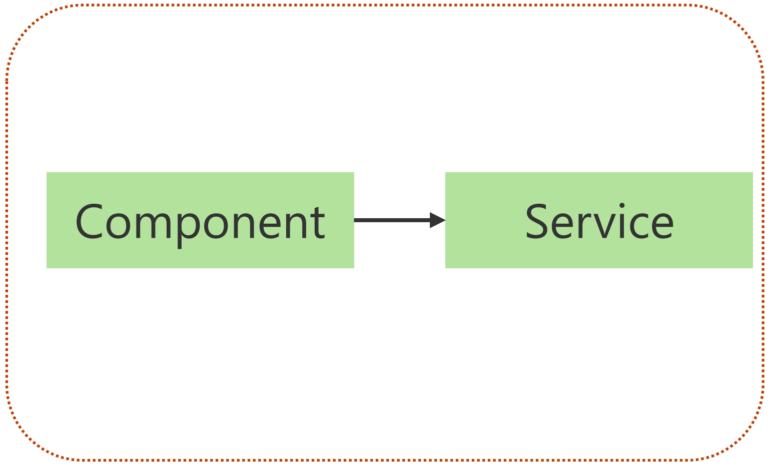
Testing in Angular.

Unit/Integration tests, mocks, spies, and more

Types of testing







End To End Testing

Test the full running app

- Live server
- Live database
- Live frontend
- Tests done by automating browser clicks, navigations, etc…

Unit Testing

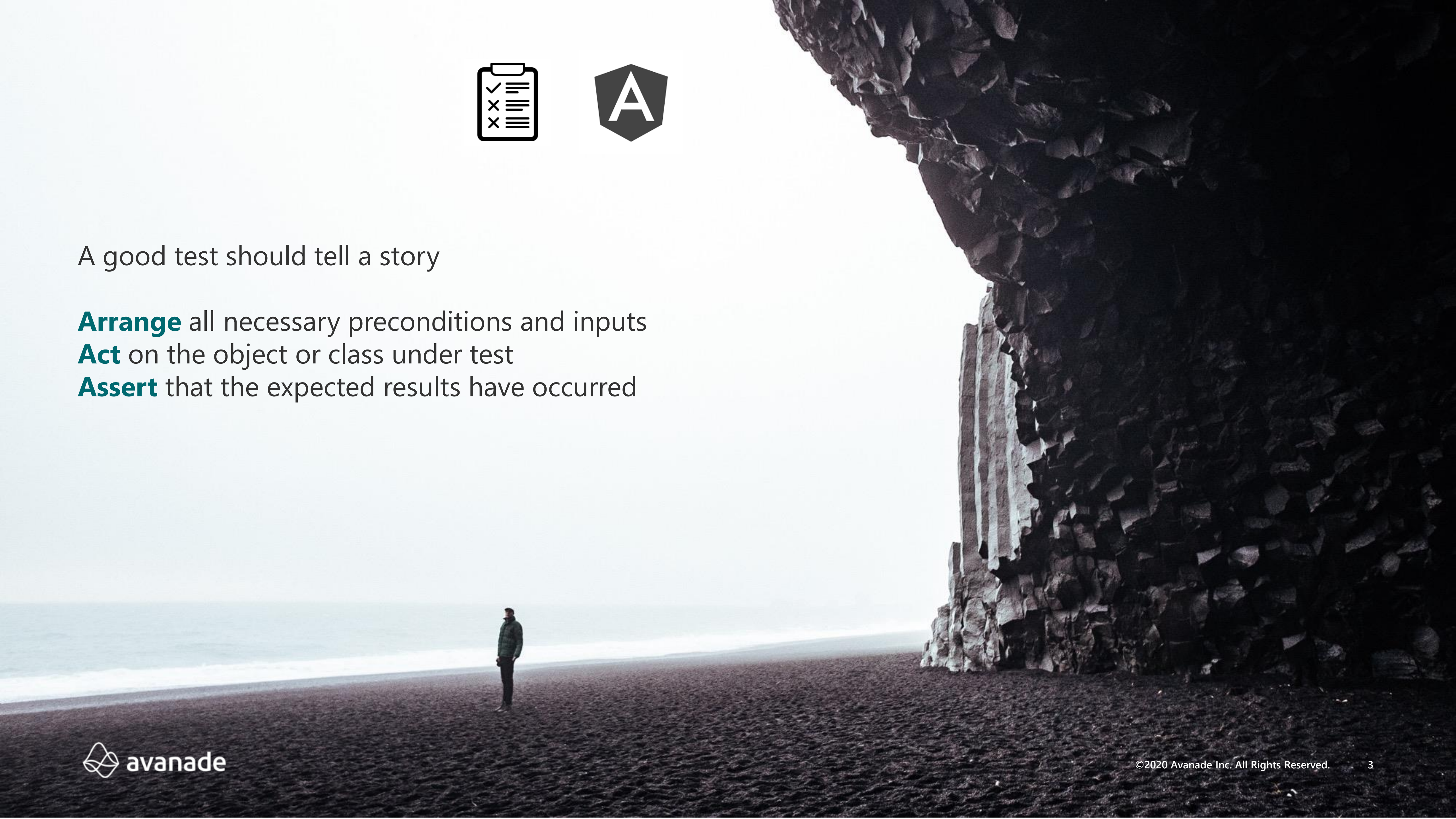
Test single unit of code

- Definition of Unit can be gray area, but usually confined to something like a class
- Mock/hide things outside of that specific unit

Integration and Functional Testing

More than a unit, less than a full running app

- Testing more than one unit
- Test interaction between subset of units



A good test should tell a story

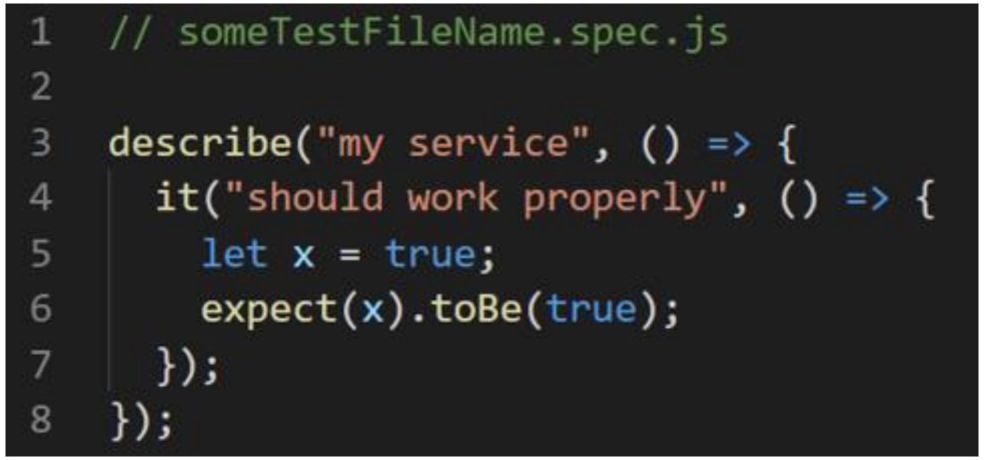
Arrange all necessary preconditions and inputs
Act on the object or class under test
Assert that the expected results have occurred

Tools We Will Use

The Angular CLI provides you with the testing tools you need

- Karma
  - The test runner
  - executes tests in a browser
- Jasmine
  - Tool for creating mocks
  - Expectations and assertions

Running Your Tests

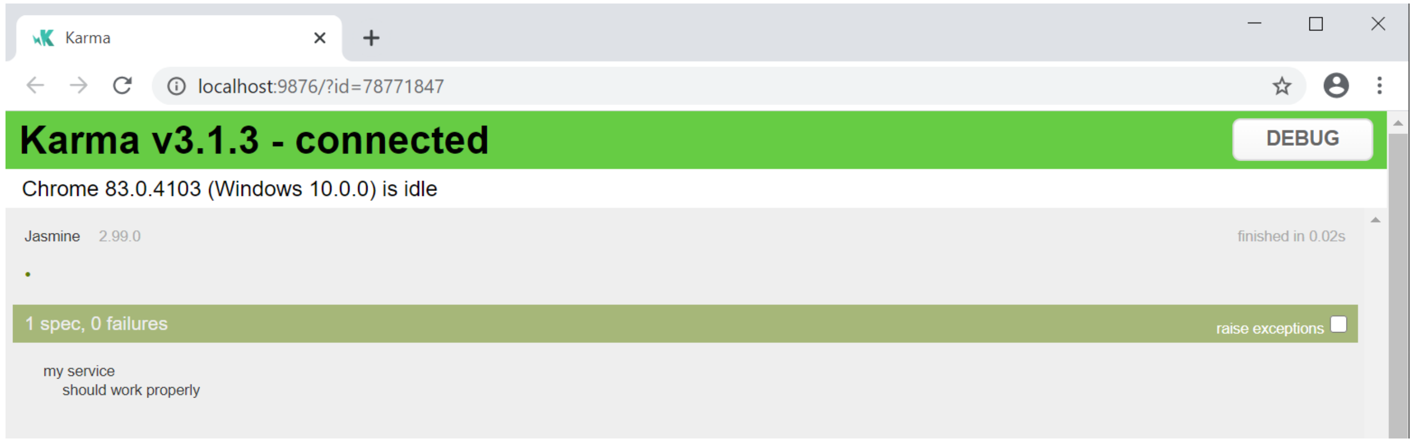
1 

```
// someTestFileName.spec.js

describe("my service", () => {
  it("should work properly", () => {
    let x = true;
    expect(x).toBe(true);
  });
});
```

2 *In command line:*

npm test

3 

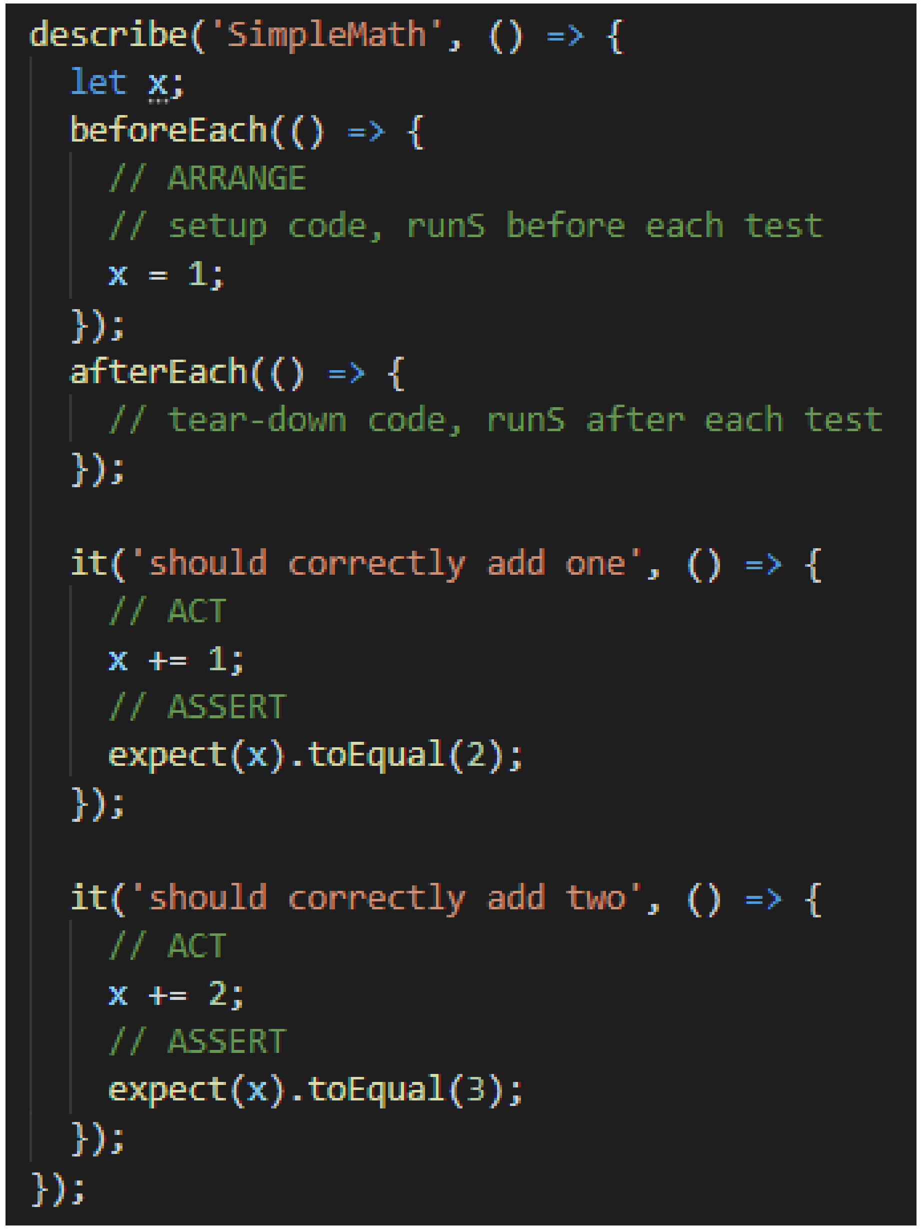
```
Karma
localhost:9876/?id=78771847
Karma v3.1.3 - connected
Chrome 83.0.4103 (Windows 10.0.0) is idle
Jasmine 2.99.0                                  finished in 0.02s
.
1 spec, 0 failures                              raise exceptions
  my service
    should work properly
```

1. Test file should end with .spec.ts
2. Run test with **`npm test`**
3. Karma will run tests

What does a test look like?

describe: indicates a test suite, which is a grouping of tests

it: indicates an individual test



```
describe('SimpleMath', () => {
  let x;
  beforeEach(() => {
    // ARRANGE
    // setup code, runS before each test
    x = 1;
  });
  afterEach(() => {
    // tear-down code, runS after each test
  });

  it('should correctly add one', () => {
    // ACT
    x += 1;
    // ASSERT
    expect(x).toEqual(2);
  });

  it('should correctly add two', () => {
    // ACT
    x += 2;
    // ASSERT
    expect(x).toEqual(3);
  });
});
```

avanade

Common Jasmine Methods

Function	Defintion
`beforeAll(function, timeout)`	Run some shared setup once before all of the specs in the describe are run.
`afterAll(function, timeout)`	Run some shared teardown once after all of the specs in the describe are run.
`beforeEach(function, timeout)`	Run some shared setup before each of the specs in the describe in which it is called.
`afterEach(function, timeout)`	Run some shared teardown after each of the specs in the describe in which it is called.
`describe(description, specDefintions)`	Create a group of specs (often called a suite). Describes can be nested to compose suite as tree
`it(description, testFunction, timeout)`	Define a single spec. Define a single spec. A spec should contain one or more expectations that test the state of the code.
`expect(actual)`	Create an expectation for a spec.
`fdescribe(desc, specs) / fit(desc, func)`	A focused describe / a focused it
`xdescribe(desc, specs) / xit(desc, func)`	A temporarily disabled describe / a temporarily disabled it
`jasmine.createSpyObj(name, methodNames[])`	returns an object that has a property for each string that is a spy.

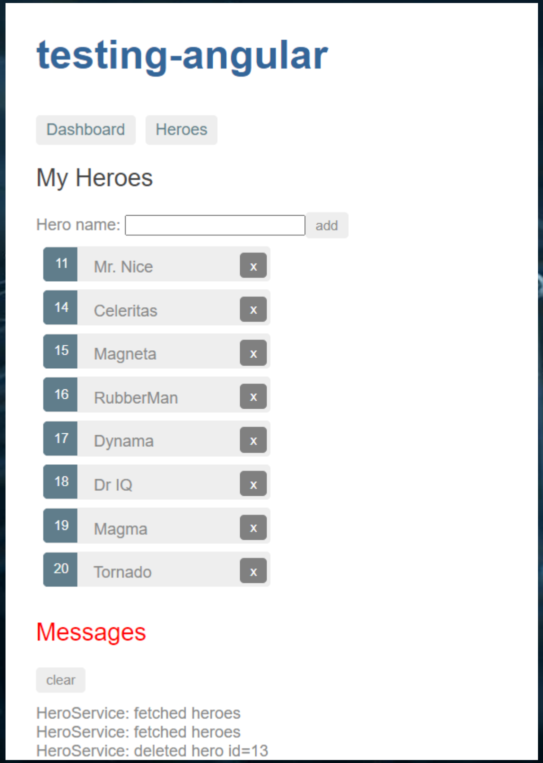
Sample App

Demo project based on Angular's intro tutorial, *Tour of Heroes*.

https://angular.io/tutorial

Let's look at the project we will be testing

```
git clone
https://github.com/SirMattCam/AngularTestingPresentation
```


testing-angular

My Heroes

Hero name: add

11	Mr. Nice
14	Celeritas
15	Magneta
16	RubberMan
17	Dynama
18	Dr IQ
19	Magma
20	Tornado

Messages

```
HeroService: fetched heroes
HeroService: fetched heroes
HeroService: deleted hero id=13
```

What we will cover

1. Isolated Unit Tests
2. Integration tests
3. Code Coverage

avanade

show me the code!

Isolated unit tests.

- Testing a service. Let’s look at message.service.spec.ts which has no dependencies

Jasmine Spies

- Use Jasmine spies to stub and track calls to functions
- jasmine.createSpyObj() creates a mock with multiple spies. It returns an object that has a property for each string in the array

```
let someService = jasmine.createSpyObj(['someFunctionA','someFunctionB','someFunctionC']);
```

- We can use createSpyObj to mock out dependencies outside our unit and make assertions on any calls to those dependencies

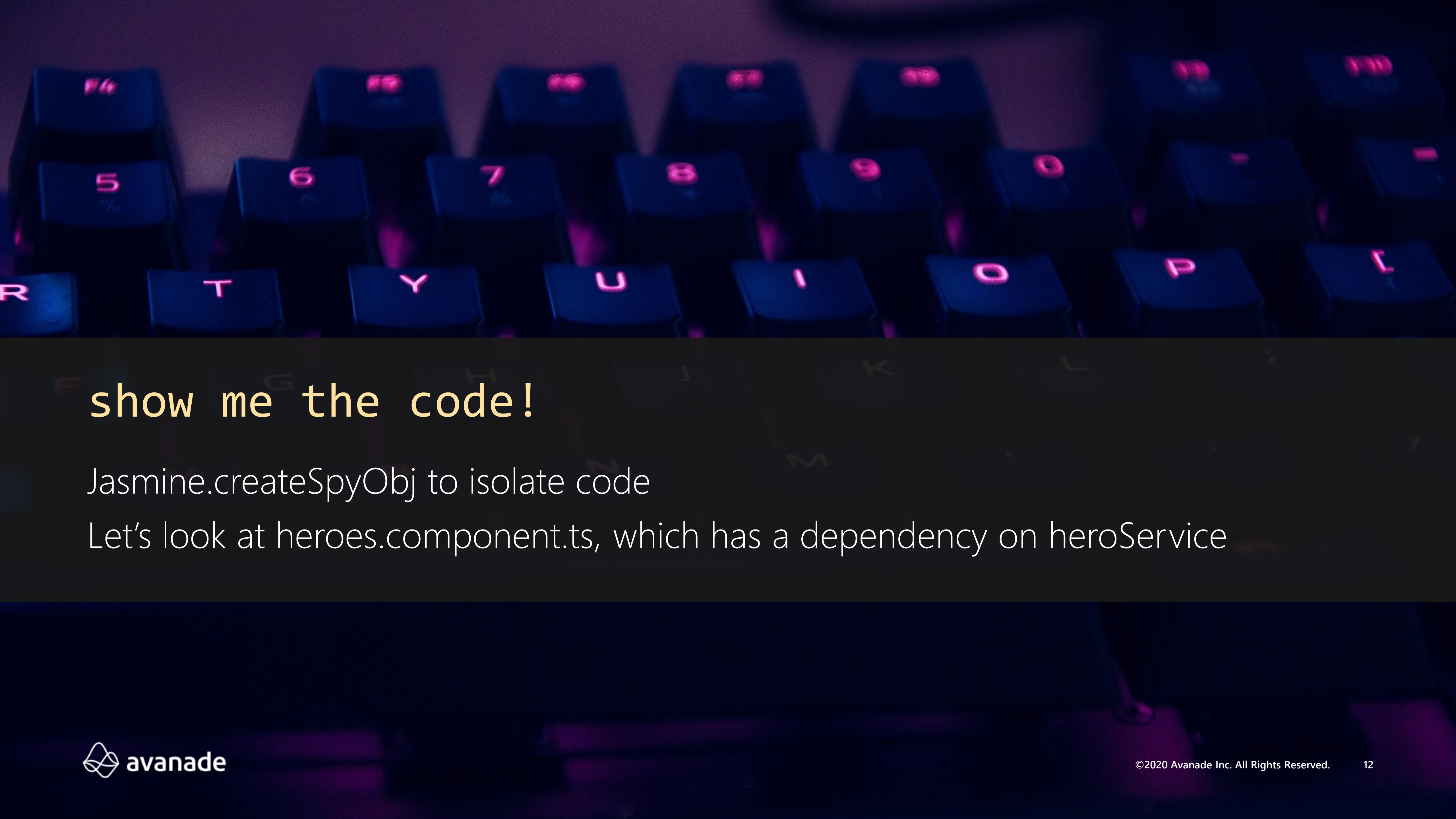
```
// Assert that someService.someFunctionA was called
expect(someService.someFunctionA).toHaveBeenCalled()
```

- If you need to actually implement the mock function, you can use `and.returnValue()`

```
someService.someFunctionA.and.returnValue('this string is returned')
```

avanade

https://jasmine.github.io/2.5/introduction

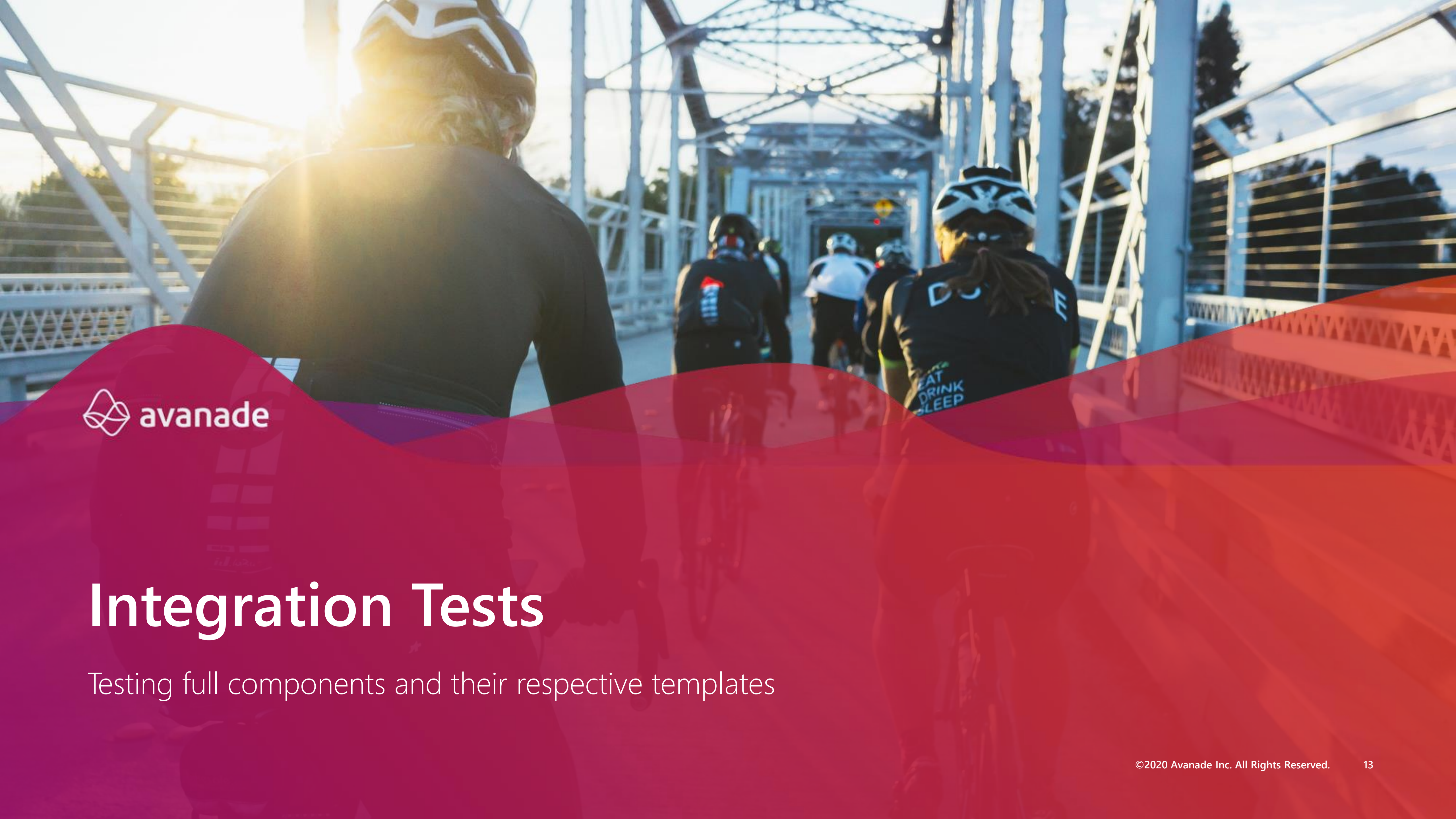


show me the code!

Jasmine.createSpyObj to isolate code

Let’s look at heroes.component.ts, which has a dependency on heroService

avanade



avanade

Integration Tests

Testing full components and their respective templates

The Testbed

- TestBed is the primary api for writing unit tests for Angular applications and libraries.
- Creates a special module specifically for testing purposes
- Allows us to test component and its template running together
- ngOnInit will be called!
- Similar to ngModule, but for testing purposes.
    - It will look like ngModule, but only use the stuff you need

TestBed reference

`TestBed.configureTestingModule({…})`

The TestBed creates a module specifically for testing

```
TestBed.configureTestingModule({ declarations: [YourFancyComponent} )
```

`TestBed.createComponent()`

- Creates test harness so you can access the created component and its corresponding element
- Returns type ComponentFixture

`ComponentFixture`

- The ComponentFixture is a test harness for interacting with the created component and its corresponding element

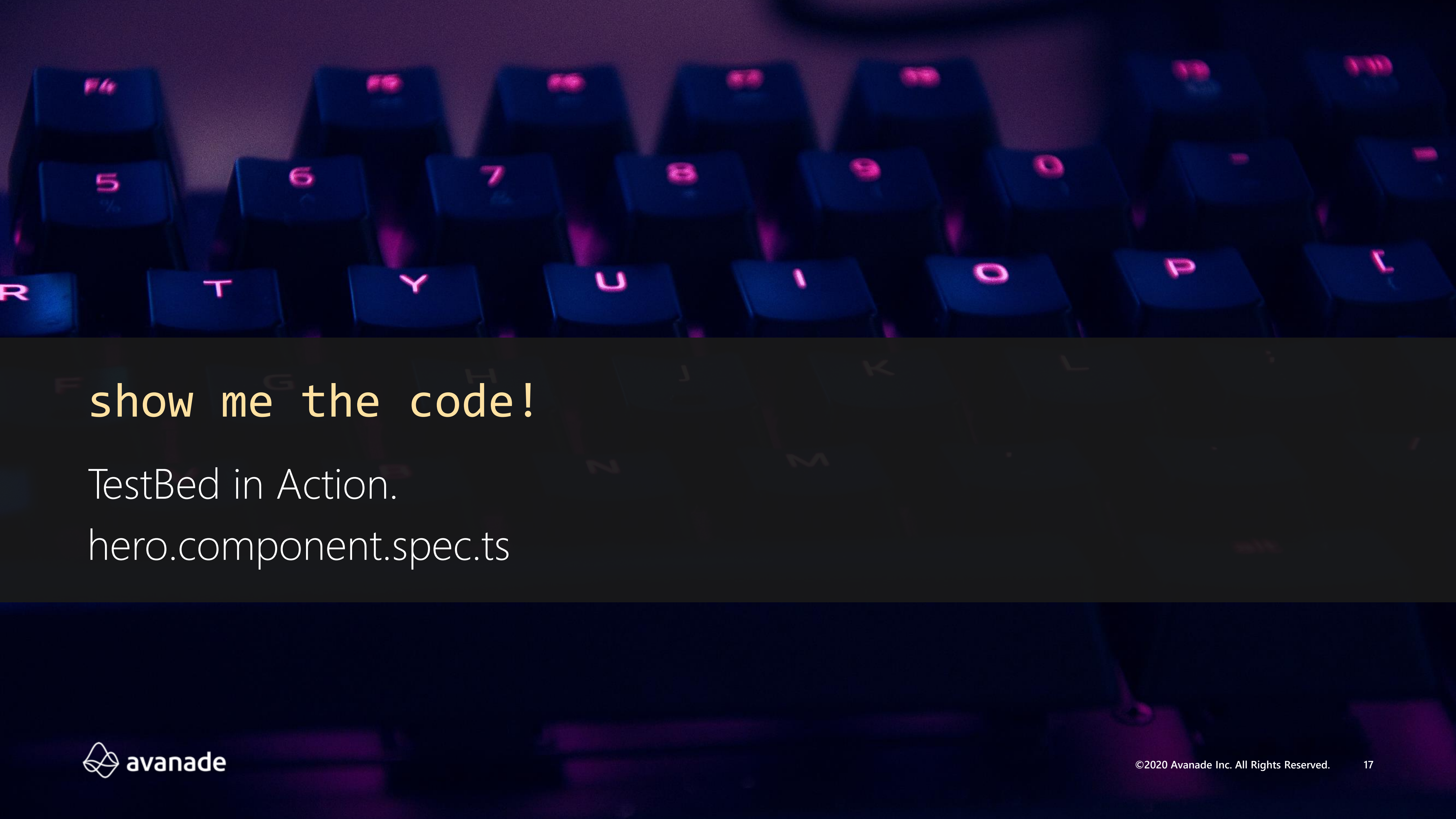
```
let fixture = TestBed.createComponent(YourFancyComponent)
```

https://angular.io/guide/testing-components-basics

ComponentFixture

```
const fixture = TestBed.createComponent(YourFancyComponent)
```

Property	Definition
`fixture.componentInstance()`	Returns actual component instance, on which you can access component's properties and methods
`fixture.nativeElement()`	Returns the HTMLElement. Can select DOM elements using querySelector(). **`Example: fixture.nativeElement.querySelector('h2').textContent`**
`fixture.debugElement()`	Wrapper around DOM node(s). Can select By.css(). Exposes additional properties. **`Example:`** **`fixture.debugElement.query(By.css('h2')).nativeElement.textContent`**
`fixture.detectChanges()`	Trigger a change detection cycle for the component.



show me the code!

TestBed in Action.

hero.component.spec.ts

avanade

Mocking Services and Child Components in TestBed

What do we do when a component has child components or service dependencies?

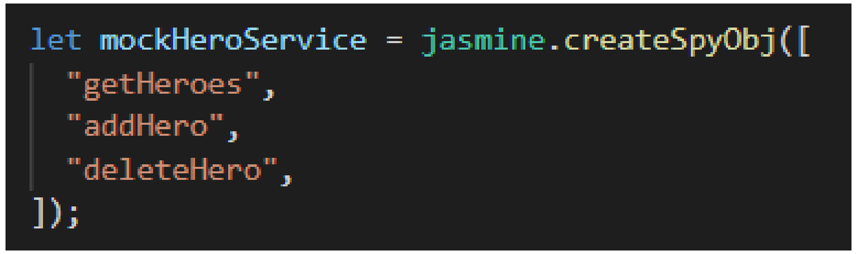
We can mock services and child components in a TestBed

Mocking Services

Many components have service dependencies.

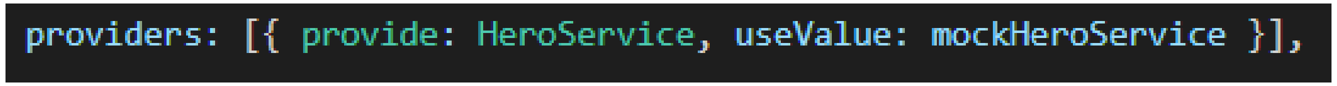
When unit testing a component, we don't care about the service(s), so we mock them.

- Use jasmine.createSpyObj() to create a fake service



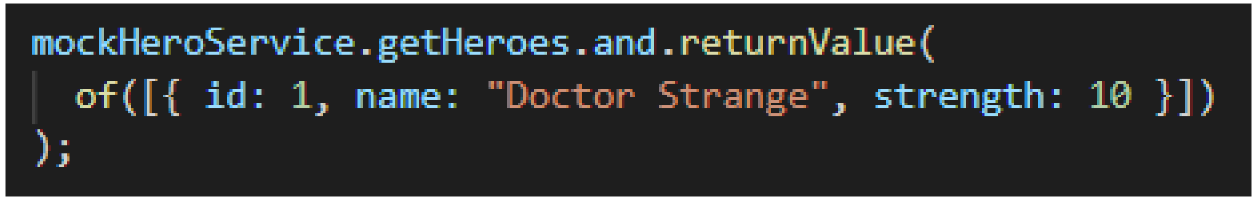
```
let mockHeroService = jasmine.createSpyObj([
  "getHeroes",
  "addHero",
  "deleteHero",
]);
```

- When someone asks for a HeroService inside this testing module, use the mock instead. *This will go in your TestBed.configureTestModule({})*



```
providers: [{ provide: HeroService, useValue: mockHeroService }],
```

- Since mockHeroService is now a Jasmine Spy Object, we can tell its methods what to return in our tests



```
mockHeroService.getHeroes.and.returnValue(
  of([{ id: 1, name: "Doctor Strange", strength: 10 }])
);
```

avanade

show me the code!

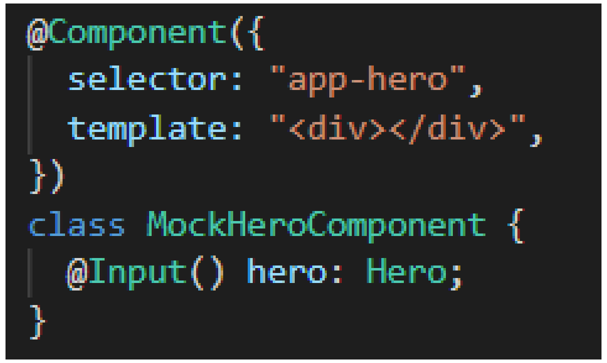
Mocking a service.

Let’s revisit heroes.component.spec.ts, and mock the service

Mocking Child Components

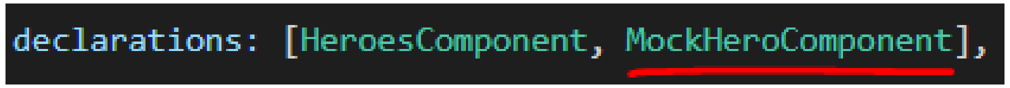
Replacing our child components with minimal placeholder components

- `schemas: [NO_ERRORS_SCHEMA]` will remove errors for undefined child components however it also will hide some glaring errors:
    - it would not fail if we misspelled an HTML element `<notanhtmltag>Hello</notanhtmltag>`
    - It's better practice to mock child components
- To mock child components,
    1. Define minimal reproduction of component



```
@Component({
  selector: "app-hero",
  template: "<div></div>",
})
class MockHeroComponent {
  @Input() hero: Hero;
}
```

    2. Include it in your declarations array of configureTestingModule



```
declarations: [HeroesComponent, MockHeroComponent],
```

avanade

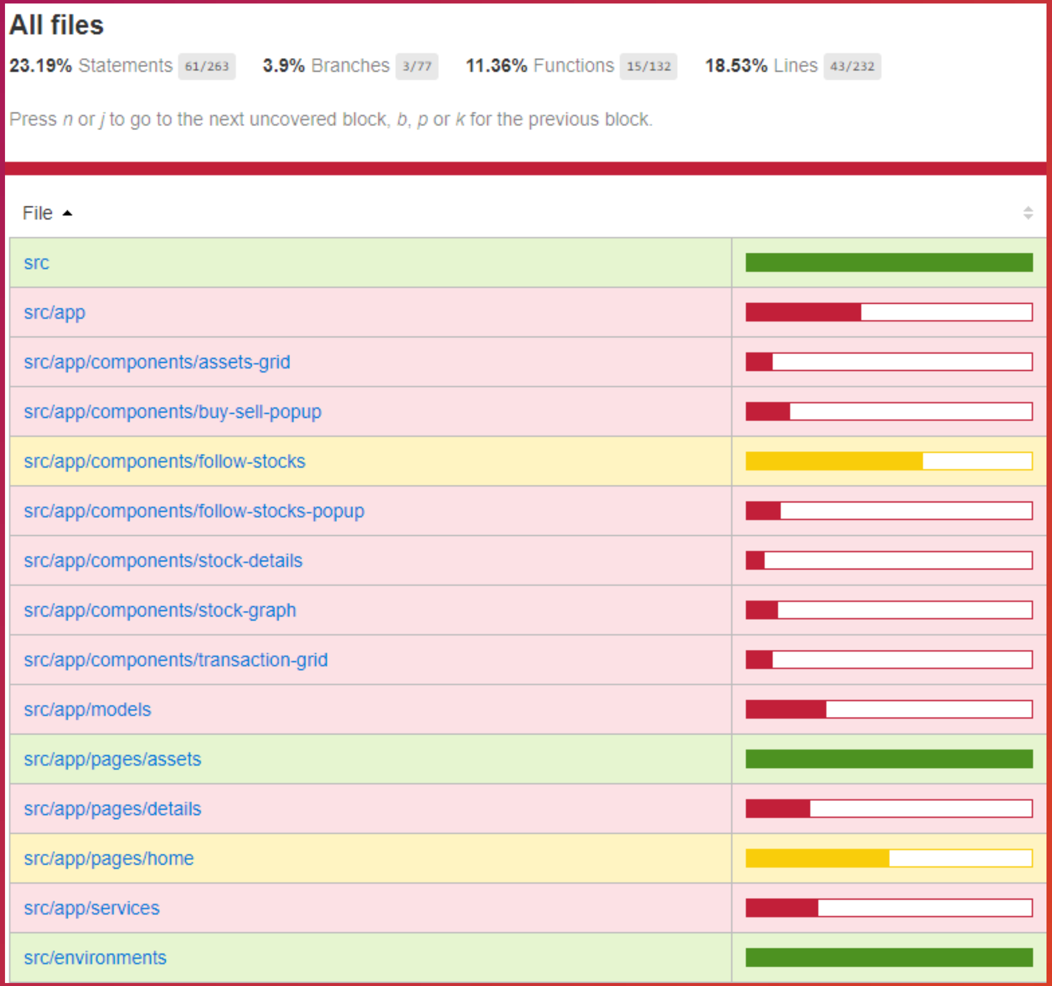
show me the code!

Mocking a child component

Revisit heroes.component.spec.ts, mocking the hero child component

avanade

Code Coverage



All files

23.19% Statements 61/263 **3.9%** Branches 3/77 **11.36%** Functions 15/132 **18.53%** Lines 43/232

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

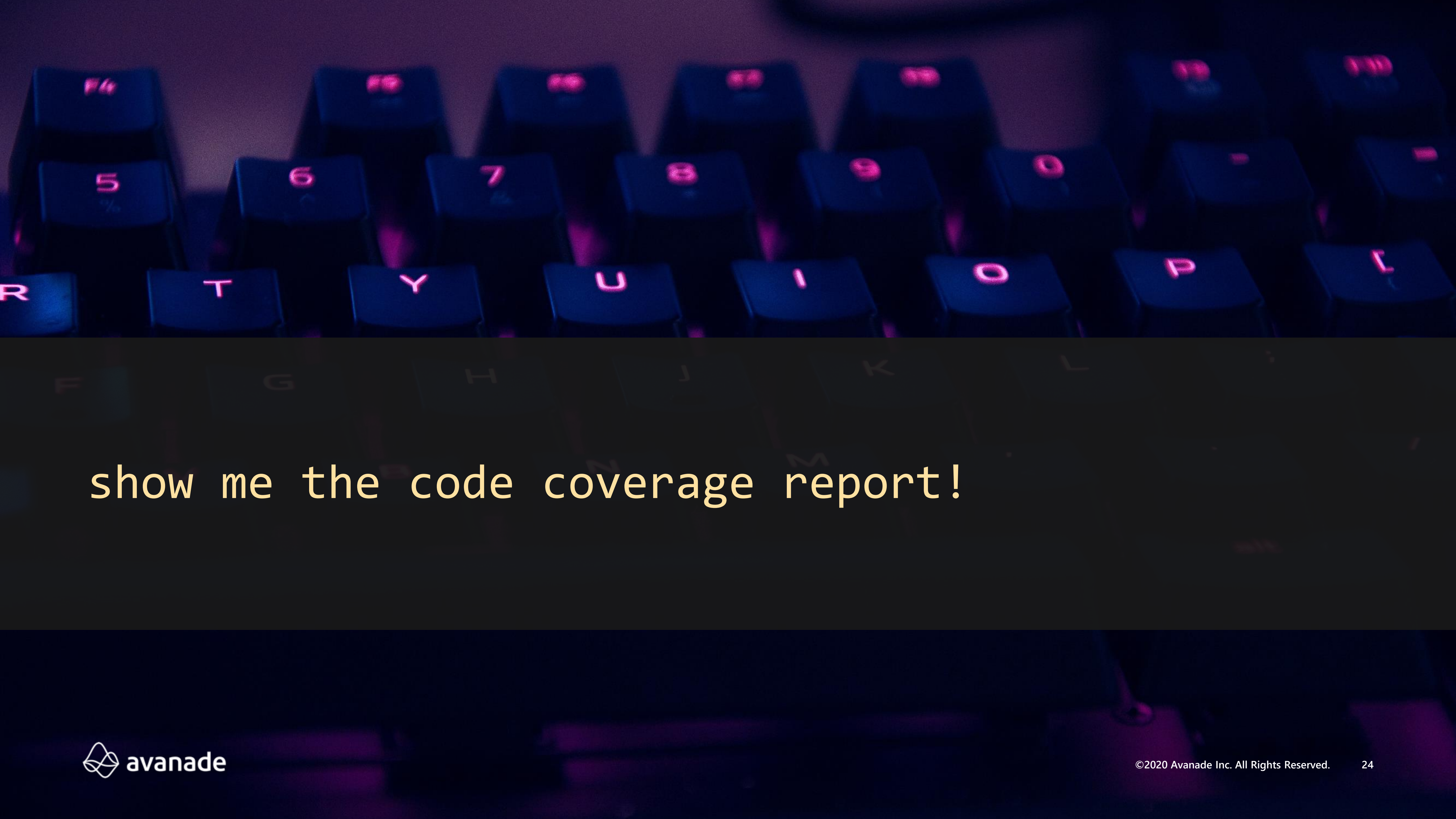
File ▲	
src	🟩
src/app	🟥
src/app/components/assets-grid	🟥
src/app/components/buy-sell-popup	🟥
src/app/components/follow-stocks	🟨
src/app/components/follow-stocks-popup	🟥
src/app/components/stock-details	🟥
src/app/components/stock-graph	🟥
src/app/components/transaction-grid	🟥
src/app/models	🟥
src/app/pages/assets	🟩
src/app/pages/details	🟥
src/app/pages/home	🟨
src/app/services	🟥
src/environments	🟩

Code coverage reports show you any parts of your code base that may not be properly tested by your unit tests.

To generate a code coverage report, run this at project root:

```
ng test --no-watch --code-coverage
```

This will create/update a coverage directory in your project root. Open coverage/index.html page to view report.



show me the code coverage report!

happy hacking

let's chat:
matthew.m.cameron@avanade.com