



# Atos

## Global Knowledge.®

# Angular Advanced Module – Testing

Peter Kassenaar –  
[info@kassenaar.com](mailto:info@kassenaar.com)

WORLDWIDE LOCATIONS

BELGIUM CANADA COLOMBIA DENMARK EGYPT FRANCE IRELAND JAPAN KOREA MALAYSIA MEXICO NETHERLANDS NORWAY QATAR  
SAUDI ARABIA SINGAPORE SPAIN SWEDEN UNITED ARAB EMIRATES UNITED KINGDOM UNITED STATES OF AMERICA

# Official documentation

FEATURES DOCS RESOURCES EVENTS BLOG test

## Testing

This guide offers tips and techniques for unit testing your Angular applications.

The guide presents tests of a sample application created in the [Tour of Heroes tutorial](#). The sample application is much like the one created in the [Tour of Heroes tutorial](#). The sample application and all tests in this guide are available for inspection and experimentation:

- [Sample app / download example](#)
- [Tests / download example](#)

## Setup

The Angular CLI downloads and installs everything you need to test an Angular application with the [Jasmine test framework](#).

The project you create with the CLI is immediately ready to test. Just run the `ng test` CLI command:

```
ng test
```

The `ng test` command builds the app in *watch mode*, and launches the [karma test runner](#).

The console output looks a bit like this:

```
10% building modules 1/1 modules 0 active
```

Not Working?

Testing

- Setup
  - Set up continuous integration
  - Configure project for Circle CI
  - Configure project for Travis CI
  - Configure CLI for CI testing in Chrome
  - Enable code coverage reports
  - Code coverage enforcement
- Service Tests
- Component Test Basics
  - Component class testing
  - Component DOM testing
- Component Test Scenarios
- Component binding
- Component with external files
- Component with a dependency
- Component with async service
- Component marble tests
- Component with inputs and outputs

<https://angular.io/guide/testing>

# Good introductory article

The screenshot shows a DZone article page. At the top, there's a navigation bar with links for REF CARDZ, GUIDES, ZONES, and various technology categories like AGILE, BIG DATA, CLOUD, DATABASE, DEVOPS, INTEGRATION, IOT, JAVA, MOBILE, PERFORMANCE, SECURITY, and WEB DEV. On the right side of the header, it says "Over a million developers have joined DZone. Sign In / Join" with a user icon, and there's a search icon.

## Testing With Angular 2: Some Recipes (Talk and Slides)

Juri Stumpflohner reflects on his recent talk about diving deeper into testing Angular 2 apps. He also links to a dedicated code repository on GitHub with the purpose of collecting testing recipes for various scenarios one might encounter while testing Angular applications.

by Juri Strumpflohner MVB · Jan. 16, 17 · Web Dev Zone

Like (-2) Comment (0) Save Tweet 4,327 Views

Join the DZone community and get the full member experience. JOIN FOR FREE

Start coding today to experience the powerful engine that drives data application's development, brought to you in partnership with Qlik.

I recently wanted to dive deeper into testing Angular applications, in specific on how to write proper unit tests for some common scenarios you might encounter.

Dave, the organizer of the Angular Hamburg Meetup group, asked me whether I'd be interested in

Subscribe ^

<https://dzone.com/articles/talk-testing-with-angular-some-recipes>

# Tooling

**Jasmine:**  
Write Unit Tests & Specs

**Karma:**  
Test Runner

**Protractor:**  
End to end testing

**Jasmine**  
Behavior-Driven Javascript

**KARMA**

On the [AngularJS](#) team, we rely on testing and we always seek better tools to make our life easier. That's why we created Karma - a test runner that fits all our needs.

[View project on GitHub](#)

[npm install karma](#)

**Testacular - JavaScript Test**

```
describe("A suite is just a function", function() {
  it("and so is a spec", function() {
    expect(true).toBe(true);
  });
});
```

[Home](#) [Quick Start](#) [Protractor Setup](#) [Protractor Tests](#) [Reference](#)

**Protractor**  
end to end testing for AngularJS

[View on GitHub](#)

[Follow @ProtractorTest](#)

Protractor is an end-to-end test framework for Angular and AngularJS applications. Protractor runs tests against your application running in a real browser, interacting with it as a user would.

**Test Like a User**  
Protractor is built on top of WebDriverJS, which uses native events and browser-specific drivers to interact with your application as a user would.

**For Angular Apps**  
Protractor supports Angular-specific locator strategies, which allows you to test Angular-specific elements without any setup effort on your part.

**Automatic Waiting**  
You no longer need to add waits and sleeps to your test. Protractor can automatically execute the next step in your test the moment the webpage finishes pending tasks, so you don't have to worry about waiting for your test and webpage to sync.

**Setup**

<https://jasmine.github.io/>

<https://karma-runner.github.io/1.0/index.html>

<http://www.protractortest.org/#/>

# An alternative to E2E-testing - Cypress

The web has evolved.  
Finally, testing has too.

Fast, easy and reliable testing for anything that runs in a browser.

\$ npm install cypress or [Download Now](#)

Install Cypress for Mac, Linux, or Windows, then [get started](#).

A screenshot of a browser window showing Cypress running tests against a social media application. The test code is visible on the left, and the application interface is on the right.

```
1 VISIT http://bootstrap-themes.github.io/a...
2 - WAIT 100
3 GET img[src="assets/img/instagram_3.jpg"]
4 - SCROLLTOVIEW(duration: 1000)
5 - CLICK
6 - WAIT 300
7 GET input[placeholder=Message]
8 - SCROLLTOVIEW(duration: 800)
```

Dave Gamache  
I wish i was a little bit taller,  
wish i was a baller, wish i  
had a girl... also.

Friends 12M | Enemies 1

Dave Gamache 4 min  
Aenean lacinia bibendum nulla sed consectetur.  
Vestibulum id ligula porta felis euismod semper. Morbi  
leo risus, porta ac consectetur ac, vestibulum at eros.  
Cras justo odio, dapibus ac facilisis in, egestas eget  
quam. Vestibulum id ligula porta felis euismod semper.  
Cum sociis natoque penatibus et magnis dis parturient  
montes, nascetur ridiculus mus.

Sponsored  
It might be time to visit

<https://www.cypress.io/>

# Headless testing ? (past: PhantomJS)

```
// Karma configuration file, see Link for more information
// https://karma-runner.github.io/0.13/config/configuration-file.html

module.exports = function (config) {
  config.set({
    ...
    reporters: ['progress', 'kjhtml'],
    port: 9876,
    colors: true,
    LogLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['ChromeHeadless'], ←
    singleRun: false
  });
};
```



# General Testing

General Jasmine syntax for testing classes and models

# Generic repo – not in ./examples !

The screenshot shows a GitHub repository page for `PeterKassenaar / ng-testing`. The repository is described as a "Sample Angular-cli project with Jasmine/Karma unit tests". It has 9 commits, 1 branch, 0 releases, and 1 contributor. The latest commit was made 8 months ago. The repository contains files like `e2e`, `src`, `.angular-cli.json`, `.editorconfig`, `.gitignore`, `README.md`, `karma.conf.js`, `package.json`, `protractor.conf.js`, and `tsconfig.json`.

File	Description	Time Ago
<code>e2e</code>	First commit of code, with sample tests.	8 months ago
<code>src</code>	some small changes	2 months ago
<code>.angular-cli.json</code>	First commit of code, with sample tests.	8 months ago
<code>.editorconfig</code>	First commit of code, with sample tests.	8 months ago
<code>.gitignore</code>	Added Firefox launcher	4 months ago
<code>README.md</code>	Added links and comment toe README.md	2 months ago
<code>karma.conf.js</code>	Adapted for ChromeHeadless testing	2 months ago
<code>package.json</code>	Adapted for ChromeHeadless testing	2 months ago
<code>protractor.conf.js</code>	First commit of code, with sample tests.	8 months ago
<code>tsconfig.json</code>	some small changes	2 months ago

[github.com/PeterKassenaar/ng-testing](https://github.com/PeterKassenaar/ng-testing)

# General testing pattern/syntax

Just a simple class and its usage:

```
import {Person} from "./person.model";
```

```
// Person.model.ts
export class Person {
    constructor(public name?: string,
                public email?: string) {
    }

    sayHello(): string {
        return `Hi, ${this.name}`;
    }
}
```

# General unit test pattern

```
// Generic testing pattern

// 1. Describe block for every test suite
describe('The Person', () => {

    // 2. Variables used by this test suite
    let aPerson;

    // 3. Setup block, run before every individual test
    beforeEach(() => {
        aPerson = new Person('Peter');
    });

    // 4. Clean up after every individual test
    afterEach(() => {
        aPerson = null;
    });

    // 5. Perform each test in an it()-block
    it('should say Hello', () => {
        let msg = aPerson.sayHello();
        expect(msg).toBe('Hi, Peter');
    });

    // 6. More it()-blocks...
});

});
```

## **So a \*.spec.ts file typically contains:**

One or more `describe()` blocks

One or more `beforeEach()` blocks

One or (typically) more `it()` blocks, using  
`expect()` statements and Jasmine *matchers*

# We're using @angular/cli here

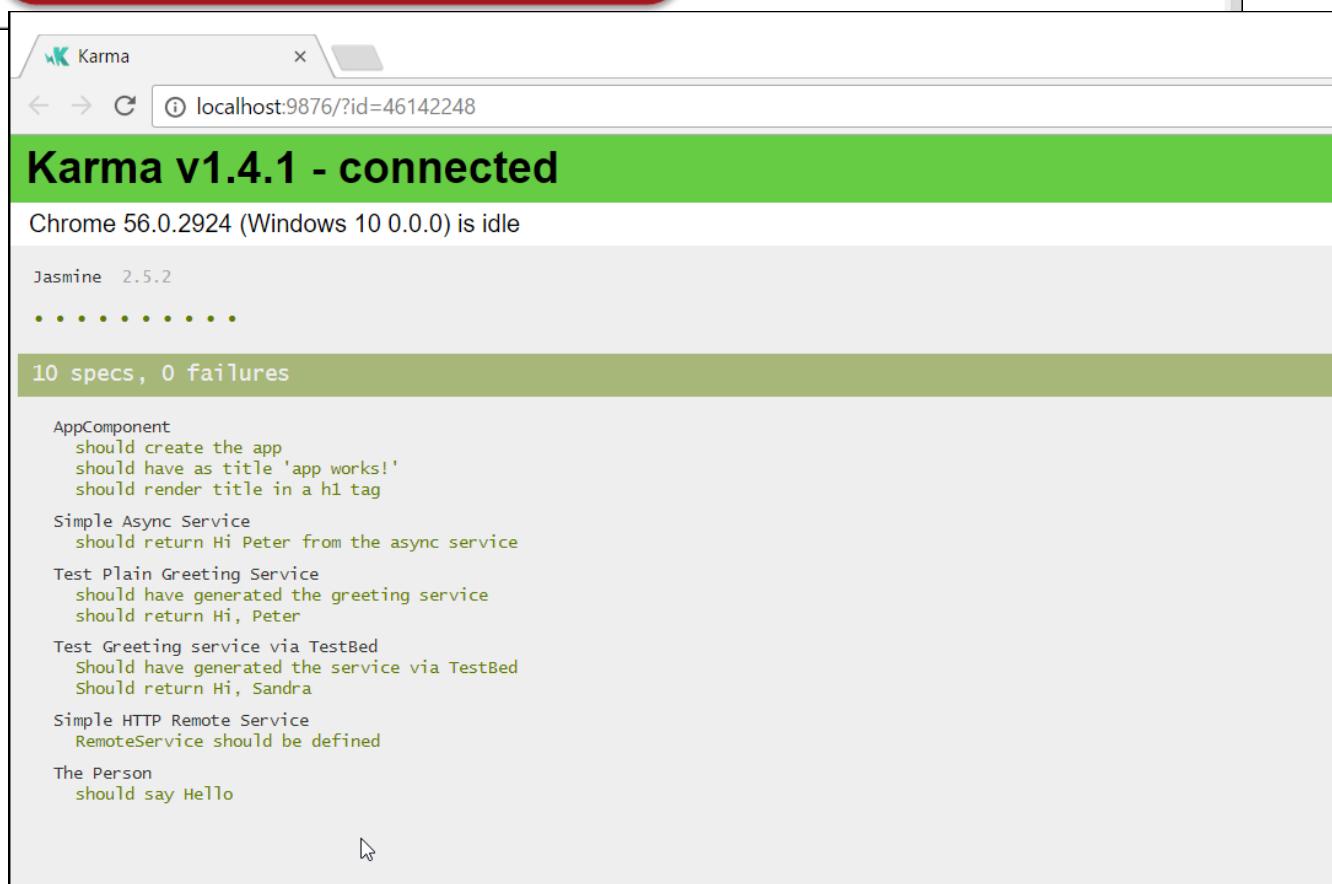
- Angular-cli: all dependencies already installed and configured.
  - Command: `ng test`
- Manual project? Install & setup karma and jasmine yourself
  - Create and adapt `karma.conf.js`
  - `karma --init`
  - Install and setup jasmine reporters
  - *We're not doing that here.*
  - See documentation



```
...  
},  
"devDependencies": {  
...  
  "@types/jasmine": "2.5.38",  
  "@types/node": "~6.0.60",  
  "codelyzer": "~2.0.0",  
  "jasmine-core": "~2.5.2",  
  "jasmine-spec-reporter": "~3.2.0",  
  "karma": "~1.4.1",  
  "karma-chrome-launcher": "~2.0.0",  
  "karma-cli": "~1.0.1",  
  "karma-jasmine": "~1.1.0",  
  "karma-jasmine-html-reporter": "^0.2.2",  
  "karma-coverage-istanbul-reporter": "^0.2.0",  
  "protractor": "~5.1.0",  
...  
}  
...
```

# Browser like...

```
C:\Users\Peter Kassenaar\Desktop\ng-testing>ng test
10 03 2017 16:50:40.157:WARN [karma]: No captured browser, open http://localhost:9876/
10 03 2017 16:50:40.170:INFO [karma]: Karma v1.4.1 server started at http://0.0.0.0:9876/
10 03 2017 16:50:40.171:INFO [launcher]: Launching browser Chrome with unlimited concurrency
10 03 2017 16:50:40.310:INFO [launcher]: Starting browser Chrome
10 03 2017 16:50:42.220:INFO [Chrome 56.0.2924 (Windows 10 0.0.0)]: Connected on socket 30x3NIH4ohFYk
MAuAAAA with id 46142248
Chrome 56.0.2924 (Windows 10 0.0.0): Executed 10 of 10 SUCCESS (0.461 secs / 0.439 secs)
```



# On Angular-specific terms

- TestBed
- inject
- async
- fakeAsync
- ComponentFixture
- DebugElement
- configureTestingModule

# Basic component test included

```
1 import { TestBed, async } from '@angular/core/testing';
2 import { AppComponent } from './app.component';
3
4 describe('AppComponent', () => {
5   beforeEach(async(() => {
6     TestBed.configureTestingModule({
7       declarations: [
8         AppComponent
9       ],
10    }).compileComponents();
11  }));
12
13 it('should create the app', () => {
14   const fixture = TestBed.createComponent(AppComponent);
15   const app = fixture.debugElement.componentInstance;
16   expect(app).toBeTruthy();
17 });
18
19 it(`should have as title 'oceTestApp'`, () =>{
20   const fixture = TestBed.createComponent(AppComponent);
21   const app = fixture.debugElement.componentInstance;
22   expect(app.title).toEqual('oceTestApp');
23 });
24
25 // + 'should render title in a h1 tag' () -> f
```

# Breaking the .spec file down:

- TestBed – the Angular Testing implementation of a Module
- .configureTestingModule() – configure only the parts of the module you want to test
- .compileComponents() – we’re testing a component here. Not necessary for services, models, etc.
- fixture – is of type ComponentFixture, acts as a wrapper around the actual component
- debugElement.nativeElement - access to the DOM of the actual component instance
- detectChanges – run change detection manually

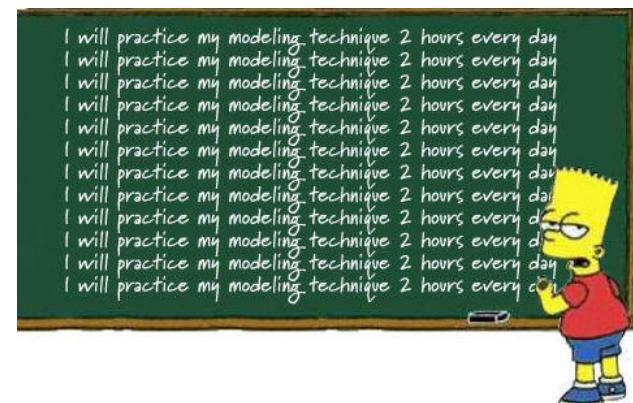
# Running the tests

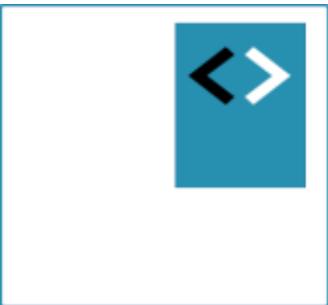
- ng test
  - Run all the tests in the .spec-files in the project
  - You can keep this running in the background!
    - It might slow down development
- fit - run this test only
- xit - run all tests except this one



# Workshop

- Create a new, empty Angular Project
- Run `ng test` – identify what tests are run
- Try to make some simple changes to the tests. See if they still run
  
- Add a new component and run its tests
- Add an array of Cities and functionality for adding and deleting a city
- Write tests for the new component.
  - Add a test for the `counter` property
- <https://github.com/PeterKassenaar/ng-testing>
- Example: `.../spy/spy.component.spec.ts`





# Testing Services

# Testing a single service

One of the more easier concepts to test. So let's start there.

```
// greeting.service.ts

import {Injectable} from '@angular/core';

@Injectable()
export class GreetingService {

  constructor() {
  }

  greet(name: string): string {
    return `Hi, ${name}`;
  }
}
```

```
// greeting.service.spec.ts

import {GreetingService} from './greeting.service';
describe('Test Plain Greeting Service', () => {
  let greetingService;
  beforeEach(() => {
    greetingService = new GreetingService();
  });

  it('should have generated the greeting service', () => {
    expect(greetingService).toBeTruthy()
  });

  it('should return Hi, Peter', () => {
    let msg = greetingService.greet('Peter');
    expect(msg).toEqual('Hi, Peter');
  });
});
```

# Output

```
Terminal
+ START:
✗ Test Plain Greeting Service
    ✓ should have generated the greeting service
    ✓ should return Hi, Peter
The Person
    ✓ should say Hello
```

# But what about DI?

- Most of the time we don't have a simple , single service
- We use it in the context of an `ngModule()`

```
...
import {GreetingService} from './shared/services/01-greeting.service';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [GreetingService], ←
  bootstrap: [AppComponent]
})
export class AppModule { }
```

# Using TestBed

- In testing we have the same concept of modules, using TestBed
- TestBed has a method `.configureTestingModule()` to mimic an NgModule.
- You only specify the stuff you need!
  - No need to build the complete module, importing all dependencies

```
let service;  
beforeEach(() => {  
  TestBed.configureTestingModule({  
    providers: [GreetingService]  
  });  
  service = TestBed.get(GreetingService);  
});
```

Dependencies

Get instance of the service  
via `TestBed.get()`

*“**TestBed** is the primary api for writing unit tests for Angular applications and libraries.”*

The screenshot shows the Angular API documentation for the `TestBed` class. The left sidebar has sections for Getting Started, Tutorial, Fundamentals, Techniques, and API, with `stable (v4.3.4)` selected. The main content area shows the `TestBed` class with tabs for CLASS and DOCUMENTATION. The CLASS tab displays information about the npm package (@angular/core), the module (import { TestBed } from '@angular/core/testing';), and the source file (core/testing/src/test\_bed.ts). A description below states: "Configures and initializes environment for unit testing and provides methods for creating components and services in unit tests." The DOCUMENTATION tab contains an overview and a code block showing the implementation of the `TestBed` class.

TestBed CLASS

npm Package	@angular/core
Module	import { TestBed } from '@angular/core/testing';
Source	core/testing/src/test_bed.ts

Configures and initializes environment for unit testing and provides methods for creating components and services in unit tests.

### Overview

```
1. class TestBed implements Injector {  
2.   static initTestEnvironment(ngModule: Type<any>|Type<any>[], platform: PlatformRef, aotSummaries?: () => any[]): TestBed  
3.   static resetTestEnvironment()  
4.   static resetTestingModule(): typeof TestBed  
5.   static configureCompiler(config: {providers?: any[]; useJit?: boolean;}): typeof TestBed  
6.   static configureTestingModule(moduleDef: TestModuleMetadata): typeof TestBed  
7.   static compileComponents(): Promise<any>  
8.   static overrideModule(ngModule: Type<any>, override: MetadataOverride<NgModule>): typeof TestBed  
9.   static overrideComponent(component: Type<any>, override: MetadataOverride<Component>): typeof TestBed  
10.  static overrideDirective(directive: Type<any>, override: MetadataOverride<Directive>): typeof TestBed
```

<https://angular.io/api/core/testing/TestBed>

# Async behavior

- If the services uses async calls, for instance Promises or Observables
- The spec-file will always returns true, because the `expect`-statement is not run in the `.then()`-clause
- So this wil NOT work:

```
it('should return Hi Peter from the async service', ()=>{
  service.greetAsync('Peter')
    .then(result =>{
      expect(result).toEqual('Hi, Petertest');
    })
});
```



Test will incorrectly pass

# Using Angular `async` and `fakeAsync`

- Solution: import and use Angular `async()` or `fakeAsync()` construct and wrap the expect statement in this function
- Jasmine will wait for the `async()` function to return and *then* perform the test

Use `async()`. Test will run OK

```
it('should return Hi Peter from the async service', async(()=>{
  service.greetAsync('Peter')
    .then((result)=>{
      expect(result).toEqual('Hi, Peter');
    })
}))
```

# Async vs. fakeAsync

- Mostly interchangeable
- fakeAsync offers more configuration/control, but is used less often
- Use fakeAsync if you need manual control of zones tick() function



Google

```
it('should work with fakeAsync', fakeAsync(() => {
  let value;
  service.simpleAsync().then((result) => {
    value = result;
  });
  expect(value).not.toBeDefined();

  tick(50);
  expect(value).not.toBeDefined();

  tick(50);
  expect(value).toBeDefined();
}));
```

“fakeAsync() makes your async code synchronous”

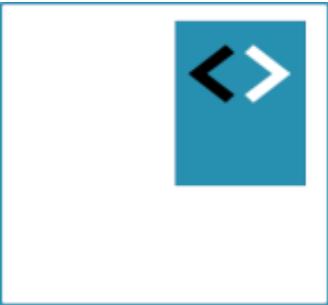
<https://stackoverflow.com/questions/42971537/what-is-the-difference-between-fakeasync-and-async-in-angular2-testing>

# Async documentation

The screenshot shows the Angular documentation website. The top navigation bar includes links for FEATURERS, DOCS, RESOURCES, EVENTS, and BLOG, along with a search bar. On the left, a sidebar menu under 'GETTING STARTED' lists TUTORIAL, FUNDAMENTALS, TECHNIQUES, and API, with 'stable (v4.3.4)' selected. The main content area is titled 'async' and shows it as a 'FUNCTION'. Below this, there's a table with three rows: 'npm Package' (@angular/core), 'Module' (import { async } from '@angular/core/testing';), and 'Source' (core/testing/src/async.ts). A code snippet for the function signature is provided: `function async(fn: Function): (done: any) => any;`. The 'Description' section explains that it wraps a test function in an asynchronous test zone. An 'Example:' section contains sample code:

```
it('...', async(inject([AClass], (object) => {
  object.doSomething.then(() => {
    expect(...);
  })
}));
```

<https://angular.io/api/core/testing/async>



# Mocking backend

Testing asynchrounous XHR-calls

*Like any external dependency, the HTTP backend needs to be mocked so your tests can simulate interaction with a remote server.*

*The @angular/common/[http](#)/testing library makes setting up such mocking straightforward.*

# Testing XHR calls

Setup (dummy) remote service to fetch some data over HTTP

```
constructor(private http: HttpClient) {  
}  
// Get fake people  
public getPeople(): Observable<Person[]> {  
    return this.http  
        .get('someEndPoint/somePeople.json')  
        .map(result => result.json());  
}
```

# Using HttpClientTestingModule and HttpTestingController

- Don't perform the tests to a 'real' API with HttpClientModule
  - Import HttpClientTestingModule in your .spec-file
- Mock your backend by giving the test fake data

*"A test expects that certain requests have or have not been made, performs assertions against those requests, and finally provide responses by "flushing" each expected request."*

# Import the correct modules

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

*Then, add the HttpClientTestingModule to the TestBed and write the tests*

```
// 2. Setup in the beforeEach() block
beforeEach(() => {
  TestBed.configureTestingModule({
    imports: [HttpClientTestingModule],
    providers: [RemoteService]
});
// 2a. Assign variables.
injector = getTestBed();
remoteService = injector.get(RemoteService);
httpMock = injector.get(HttpTestingController);
// 2b. Our mocked response
mockUsers = [
  {
    name: 'Peter',
    email: 'info@kassenaar.com'
  },
  {
    name: 'Sandra',
    email: 'sandra@kassenaar.nl'
  }
]);
});
```

HttpClientTestingModule

Mocked (fake) data

*Now requests made in the tests will hit the testing backend instead of the normal backend.*

# Write test for mocked backend

```
it('should return an Observable<User[]>', () => {
  // Make an HTTP GET request
  remoteService.getPeople()
    .subscribe(users => {
      expect(users.length).toBe(2);
      expect(users).toEqual(mockUsers);
    });
  // verify and flush our request
  const req = httpMock.expectOne(remoteService.url);
  expect(req.request.method).toBe("GET"); // just to be safe. Not mandatory.

  // Only after a flush() the .subscribe expectations are evaluated and available!
  req.flush(mockUsers);

  // Finally, verify if there are no outstanding requests.
  // httpMock.verify()
});
```

Subscriber. Results are  
only available after a  
Flush

Provide the subscriber  
with our (fake) data by  
calling .flush()

# Documentation on HttpTestingModule

The screenshot shows the Angular documentation website with a blue header bar containing the Angular logo and links for FEATURES, DOCS, RESOURCES, EVENTS, and BLOG. The main content area has a sidebar on the left with a tree view of documentation categories like STARTED, FEATURES, and DEPLOYMENT. The main content on the right is titled "Testing HTTP requests". It discusses the need to mock the HTTP backend for testing and provides a "Mocking philosophy" section. A callout box contains instructions for running sample tests or downloading an example. The "Setup" section explains how to begin testing calls to HttpClient. A blue button at the bottom of this section is labeled "app/testing/http-client.spec.ts (imports)".

STARTED

FEATURES

DOCS

RESOURCES

EVENTS

BLOG

## Testing HTTP requests

Like any external dependency, the HTTP backend needs to be mocked so your tests can simulate interaction with a remote server. The `@angular/common/http/testing` library makes setting up such mocking straightforward.

### Mocking philosophy

Angular's HTTP testing library is designed for a pattern of testing wherein the app executes code and makes requests first. Then a test expects that certain requests have or have not been made, performs assertions against those requests, and finally provide responses by "flushing" each expected request. At the end, tests may verify that the app has made no unexpected requests.

You can run [these sample tests](#) / [download example](#) in a live coding environment. The tests described in this guide are in `src/testing/http-client.spec.ts`. There are also tests of an application data service that call `HttpClient` in `src/app/heroes/heroes.service.spec.ts`.

### Setup

To begin testing calls to `HttpClient`, import the `HttpClientTestingModule` and the mocking controller, `HttpTestingController`, along with the other symbols your tests require.

app/testing/http-client.spec.ts (imports)

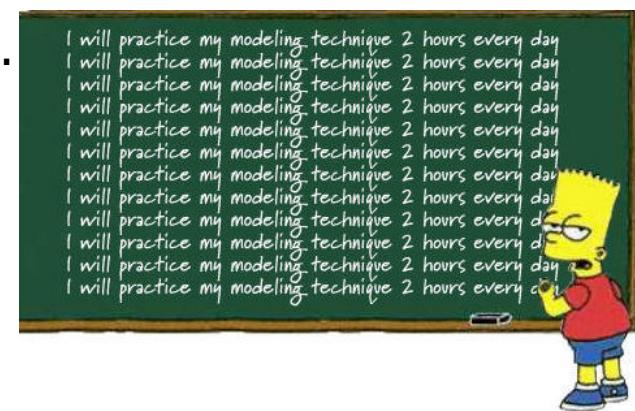
<https://angular.io/guide/http#testing-http-requests>

# Summary

- What have we learned?
  - Jasmine Syntax: `describe()`, `beforeEach()`, `it()`, `expect()`
  - `TestBed.configureTestingModule()`
  - `TestBed.get()`
  - Testing Classes
  - Testing Services
- Mocking backend
  - `HttpClientTestingModule`, `HttpTestingController`
  - `flush()`

# Workshop

- Study the example `.../ng-testing`.
- Study `shared/model/10-car.model.ts` and create a test suite for it
- Study `10-car.service.ts` and create a test suite for it, using `TestBed.configureTestingModuleTestingModule()`
- Study `11-car.remote.service.ts` and create a test suite for it, using `HttpClientTestingModule`
  - The service is fetching Cars from a dummy backend.
  - Also write a test for the second method, fetching cars from a specific year.





# Testing components

Building blocks of every application

# Testing components

- Less often tested than services, routes, etc.
- Test View components only for their `@Input()` en `@Output()`'s
- Test Smart components as simple as possible
- New concepts:
  - `.compileComponents()`
  - `Fixture`
  - `.detectChanges()`
  - `componentInstance`
  - `DebugElement`
  - `NativeElement`

# Simple Component testing

- Generate a simple component, using `{} ... {}` data binding for instance
- Create a TestBed, with instance of the component
  - Now using the declarations array
- Compile the component using `.compileComponents()`.

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

*“Do not configure the `TestBed` after calling `compileComponents`. Make `compileComponents` the last step before calling `TestBed.createComponent` to instantiate the component-under-test.”*

# Component Instance

- Component is now compiled for testing purposes (and added to the TestBed), but is not instantiated yet
- Use variables `component` and `fixture` for that.

```
let component: CityComponent;
let fixture: ComponentFixture<CityComponent>;  
  
beforeEach(async(() => {  
    ...  
    fixture = TestBed.createComponent(CityComponent);  
    component = fixture.componentInstance;  
    fixture.detectChanges();  
})  
);
```

*“The fixture provides access to the component instance itself and to the [DebugElement](#), which is a handle on the component's DOM element.”*

# Complete test

```
import {async, ComponentFixture, TestBed} from '@angular/core/testing';

import {CityComponent} from './city.component';

describe('CityComponent', () => {
  let component: CityComponent;
  let fixture: ComponentFixture<CityComponent>;

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

    fixture = TestBed.createComponent(CityComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  })
);

  it('should be created', () => {
    expect(component).toBeTruthy();
  });
});
```

# Testing a method on a component

Suppose the component has a `setCity(name)` method like so:

```
setCity(name: string) {  
  this.city = name;  
}
```

Usage in a test

```
it('city should have the name Amsterdam', ()=>{  
  component.setCity('Amsterdam');  
  expect(component.city).toEqual('Amsterdam');  
});
```

# Accessing DOM-elements

Handy Helper library:<sup>By</sup>

- Provides querying the DOM like jQuery and .querySelector[All]

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

Use debugElement and NativeElement.

Don't forget to trigger change detection on the fixture!

```
it('should have rendered Amsterdam on the page', ()=>{
  const de = fixture.debugElement.query(By.css('h1'));
  const element = de.nativeElement;
  component.setCity('Amsterdam');
  fixture.detectChanges();
  expect(element.textContent).toContain('Amsterdam');
});
```



# Testing @Input() and @Output

Testing component attributes and events,  
using Jasmine spies

# Testing @Input() parameters

- An Input()-parameter is just a property on the class.
- So treat it as such...

```
import {Component, Input} from '@angular/core';

@Component({
  selector    : 'app-input',
  templateUrl: './input.component.html',
  styles      : []
})
export class InputComponent {
  @Input() msg: string;
}
```

With the template just: "{{ msg }}"

# Writing the @Input test

```
import {async, ComponentFixture, TestBed} from '@angular/core/testing';

import {InputComponent} from './input.component';

describe('InputComponent', () => {
  let component: InputComponent;
  let fixture: ComponentFixture<InputComponent>;

  beforeEach(async(() => {
    ...
  }));

  it('should have the message defined', ()=>{
    expect(fixture.debugElement.nativeElement.innerHTML).toEqual('');

    // now let's set the message
    component.msg = 'Hi, there';

    fixture.detectChanges();

    expect(fixture.debugElement.nativeElement.innerHTML).toEqual('Hi, there');

  })
});
```

# Different ways to access the underlying DOM

- Access DOM-element via `debugElement` and `By`:
  - `fixture.debugElement.query(By.css('h1'));`
  - Recommended. DOM is abstracted by Angular. Works also in Non-DOM environments (like server-apps)
- Access native element directly:
  - Not recommended, but certainly possible (if you \*know\*, your app will never run outside a browser environment)
  - `fixture.nativeElement.querySelector('h1');`

# Testing @Output() events

- Simple class, outputting a msg event, submitting a message.

```
import {Component, EventEmitter, Output} from '@angular/core';

@Component({
  selector    : 'app-output',
  templateUrl: './output.component.html'
})
export class OutputComponent {

  @Output() msg: EventEmitter<string> = new EventEmitter<string>();

  sendMsg(msg: string) {
    this.msg.emit(msg);
  }
}

<p>
  <button (click)="sendMsg('Hi, there')">Send Message</button>
</p>
```

# Different strategies for testing events

*1st strategy:*

simply subscribe to the event and call the method that fires the event

```
it('should fire the msg event', ()=>{
    // 1st strategy : subscribe to the msg event
    component.msg.subscribe(msg =>{
        expect(msg).toBe('Hi, there');
    });
    // emit the actual event with a value
    component.sendMsg('Hi, there');
})
```

# Creating a Jasmine Spy

- Spies are Jasmine 'watchers'
- You can query the spy and expect that
  - it has been called,
  - It has not been called
  - It has been called with a specific value,
  - It has been called a specific number of times,
  - ...
- Documentation: <https://jasmine.github.io/api/2.7/global.html#spyOn>
- Cheat sheet: <https://daveceddia.com/jasmine-2-spy-cheat-sheet/>
- Tutorial: <http://www.htmlgoodies.com/html5/javascript/spy-on-javascript-methods-using-the-jasmine-testing-framework.html>

# Jasmine spyOn()

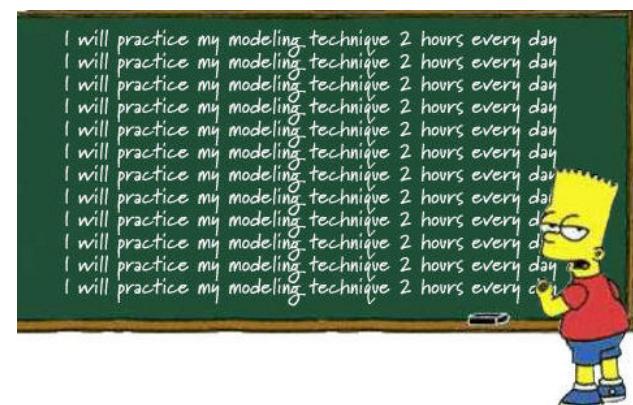
```
spyOn(Object, 'method')
```

```
it('should spy on the msg event', ()=>{
  // 2nd strategy : create a Jasmine Spy, watch the 'emit' event.
  spyOn(component.msg, 'emit');
  const button = fixture.debugElement.nativeElement.querySelector('button');
  button.click();

  expect(component.msg.emit).toHaveBeenCalledWith('Hi, there');
})
```

# Workshop

- Study car.component.ts and create a test suite for it
- Test whether the component is correctly created
- Test if this.cars[] is constructed after initialisation. Expect the length of the array to be 2.
- Test whether the @Output() event is called when clicked on a car
- Create an @Input property for the component yourself and test it





# Mocking components

Strategies for minimizing the dependency chain of components

# Multiple components

- What if an a component has multiple, nested components?
- Different possible strategies
  - Include all your components
  - Override your components at test time
  - Ignore errors and continue, using `NO_ERRORS_SCHEMA`

```
<!--card.component.html/ts/spec.ts-->
<card-header>
    Some Title
</card-header>
<card-content>
    Some content
</card-content>
<card-footer>
    Copyright (C) - 2017
</card-footer>
```

# #1 Include all components

- Import all components and reference them in declarations : [...] section
- *Pro* – complete test coverage, compile the component as it would run in the live app
- *Con* – more overhead, slower, tests for nested components are possibly elsewhere, overkill

```
import {CardComponent, CardContent, CardFooter, CardHeader} from './card.component';

describe('CardComponent', () => {
  ...
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [CardComponent, CardHeader, CardContent, CardFooter],
    })
      .compileComponents();
  }));
  ...
});
```



## #2 – Override components at test time

- Simply provide empty components to the testsuite
- *Pro* – Component would compile as at run time
- *Con* – duplicate code, not a nice view, more overhead.

```
@Component({
  selector: 'card-header',
  template: ''
})
export class CardHeader{}

@Component({
  selector: 'card-content',
  template: ''
})
export class CardContent{}

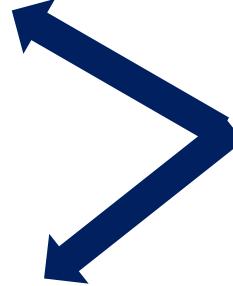
...
```

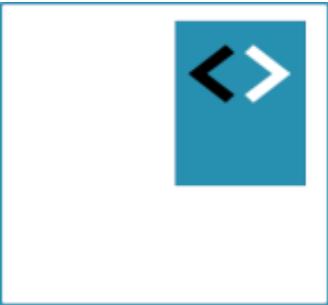
## #3 Using NO\_ERRORS\_SCHEMA

- Provide a schema to the testing module, ignoring all errors and always continue the test
- *Pro* – flexible setup, no dependencies, faster compiling
- *Con* – you might not catch other errors

```
import {NO_ERRORS_SCHEMA} from '@angular/core';

describe('CardComponent', () => {
  ...
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [CardComponent],
      schemas      : [NO_ERRORS_SCHEMA] // ignore alle errors, just go on.
    })
    .compileComponents();
    ...
  }));
  ...
});
```





# Testing @Effects

Writing unit tests for Store-enhanced applications

The screenshot shows a web browser window with the title bar "M Unit Test Your ngrx/effects in A X". The address bar contains the URL "https://netbasal.com/unit-test-your-ngrx-effects-in-angular-1bf2142dd459". The page content includes a header with a large "M" logo, a profile picture, and a "Follow" button. The main title is "Unit Test Your ngrx/effects in Angular" by "Netanel Basal" (with a small profile picture). Below the title is a snippet of Angular code:

```
@Injectable()  
export class PostsEffects {  
  constructor( private postsService : PostsService,  
    private actions$ : Actions ) {  
  }  
  
  @Effect() posts$ = this.actions$  
    .ofType(GET_POSTS)  
    .debounceTime(400)  
    .switchMap(_ => this.postsService.getPosts())  
    .map(posts => getPostsSuccess(posts))  
    .catch(error => Observable.of(getPostsFail(error)))
```

<https://netbasal.com/unit-test-your-ngrx-effects-in-angular-1bf2142dd459>

M

 Upgrade 

# Testing ngrx effects

 Adrian Fâciu [Follow](#)

Aug 30, 2017 · 8 min read

```
19  @Effect()
20  fetchDataWithState$ = this.actions$
21    .ofType(actions.DATA_WITH_STATE_FETCH)
22    .withLatestFrom(this.numberStream$, this.textStream$)
23    .map(([action, number, text]) => ({ type: actions.DATA_WITH_STATE_FETCH_SUCCESS }));
24 }
```

**Update:** Since I've wrote this, both NgRx and RxJs released some new versions that require some small changes in how we can write the tests. You can see all those in [this follow up](#).

If you decide to use a *state management* library in your [Angular](#) application (and for most of them you should :) ) a good choice is one of the [Redux](#) inspired libraries. You'll most likely have actions, reducers and pretty soon something to handle all the side effects. In case of [ngrx](#), these side effects are

<https://medium.com/@adrianfaciu/testing-ngrx-effects-3682cb5d760e>

72

M |  Angular In Depth by  Follow

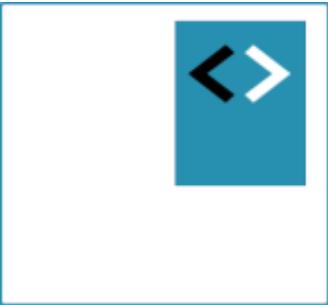
HOME ANGULAR RXJS NGRX ABOUT SUPPORT US | AG-GRID: THE BEST ANGULAR GRID IN THE WORLD

# How To Unit Test Angular Components With Fake NgRx TestStore

 Tomas Trajan [Follow](#)  
Jul 10, 2018 · 8 min read



<https://blog.angularindepth.com/how-to-unit-test-angular-components-with-fake-ngrx-teststore-f0500cc5fc26>



# End-2-End testing

Testing complete scenario's

# What is End 2 End testing

- Test complete processes, not single units
- Test in real environments, with the whole application
- Test real live situations
- Know if most important (tested) features work with the application
- Does *not* test edge cases
- Does *not* necessarily improve code quality

# E2e test Tooling

Test Automation

## Selenium



Automate browsers to perform scenario's



## Protractor

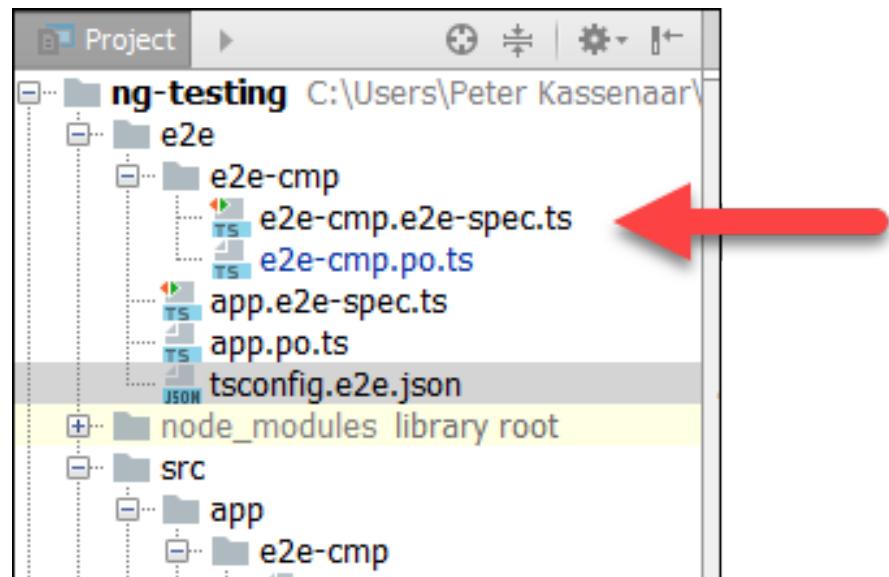
Developed by Team Angular – wrapper around selenium

# Parts of e2e tests

- Folder: /e2e
- Page Objects – describe the scenario in a \*.po.ts-file
  - Export a class with a `navigateTo()` function,
  - Plus functions for all the elements you want to retrieve/test
- Test files – write tests as usual
  - They load the Page Object class
  - Write a `beforeEach()` block to instantiate the Page Object
  - Write tests to invoke and test the elements on the page

# Executing e2e tests

- Generic command: `ng e2e`
- Examples: `\e2e-cmp`



# More information on e2e testing

- <https://coryrylan.com/blog/introduction-to-e2e-testing-with-the-angular-cli-and-protractor>

The screenshot shows a blog post by Cory Rylan. At the top, there's a navigation bar with links for ARTICLES, SPEAKING, COURSES, TRAINING, and GITHUB. Below the navigation is a profile picture of Cory Rylan, a smiling man with a beard, wearing a blue shirt with 'Angular GDE' on it. To his right, text introduces him as a Google Developer Expert and Front End Developer for VMware Clarity and Angular Boot Camp instructor, specializing in creating fast progressive web applications. A blue 'Follow @coryrylan' button is below this text. The main content area features a large Angular logo (a red hexagon with a white 'A') and the title 'Introduction to E2E Testing with the Angular CLI and Protractor'. Below the title, author information (Cory Rylan, Jan 31, 2017), publication date (Updated Feb 25, 2018 - 8 min read), and tags ('angular', 'protractor') are provided. A note at the bottom states that the article has been updated to the latest version of Angular 7. To the right of the main content, there are two boxes: one for 'ANGULAR BOOT CAMP' with a 'Sign up for Angular Boot Camp to get in person Angular training!' call-to-action, and another featuring logos for 'VJS' (Vue.js and JavaScript), 'Angular', and 'React'.

# An alternative to E2E-testing - Cypress

The web has evolved.  
Finally, testing has too.

Fast, easy and reliable testing for anything that runs in a browser.

\$ npm install cypress or [Download Now](#)

Install Cypress for Mac, Linux, or Windows, then [get started](#).

A screenshot of a browser window showing Cypress running tests against a social media application. The test code is visible on the left, and the application interface is on the right.

```
1 VISIT http://bootstrap-themes.github.io/a...
2 - WAIT 100
3 GET img[src="assets/img/instagram_3.jpg"]
4 - SCROLLTOVIEW(duration: 1000)
5 - CLICK
6 - WAIT 300
7 GET input[placeholder=Message]
8 - SCROLLTOVIEW(duration: 800)
```

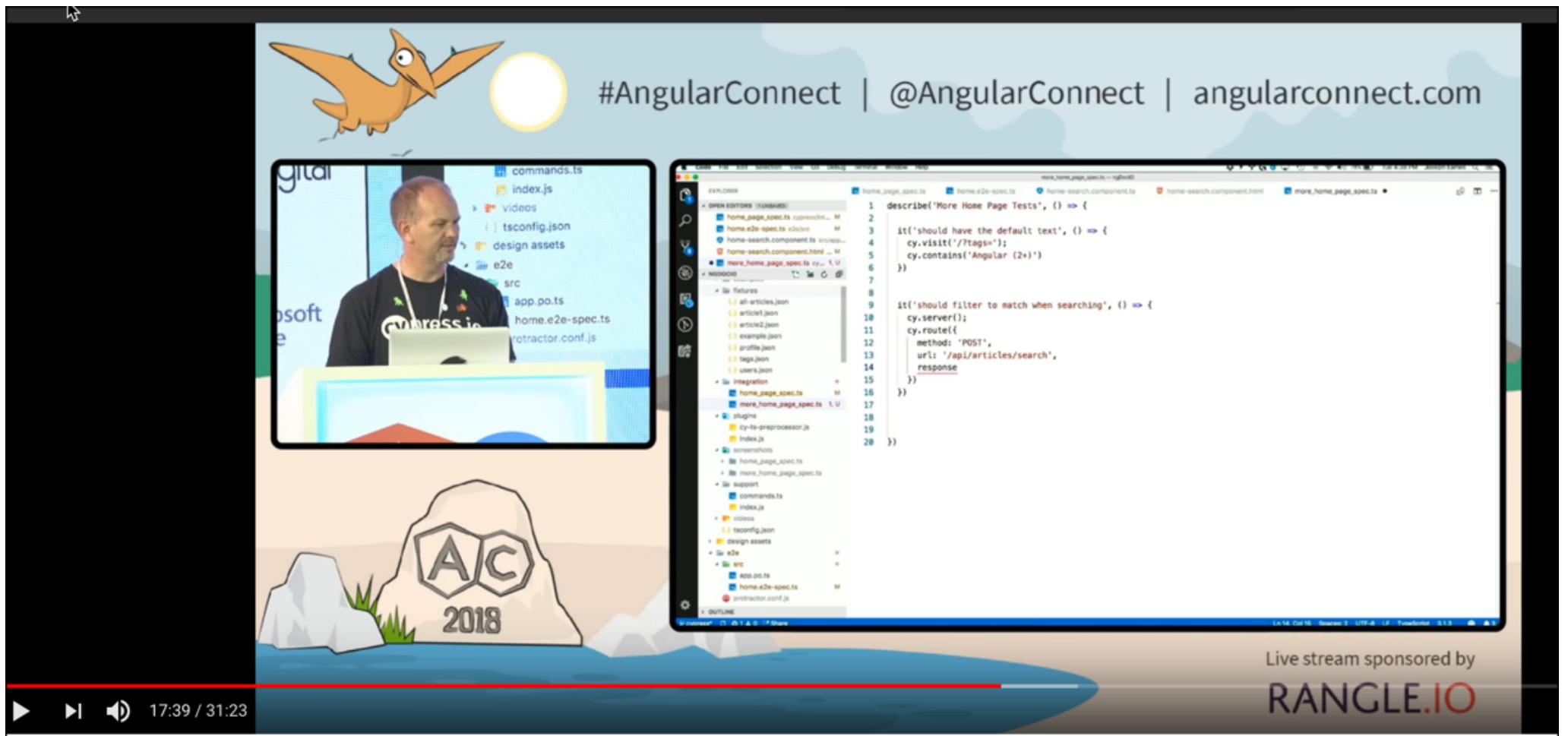
Dave Gamache  
I wish i was a little bit taller,  
wish i was a baller, wish i  
had a girl... also.

Friends 12M | Enemies 1

Dave Gamache 4 min  
Aenean lacinia bibendum nulla sed consectetur.  
Vestibulum id ligula porta felis euismod semper. Morbi  
leo risus, porta ac consectetur ac, vestibulum at eros.  
Cras justo odio, dapibus ac facilisis in, egestas eget  
quam. Vestibulum id ligula porta felis euismod semper.  
Cum sociis natoque penatibus et magnis dis parturient  
montes, nascetur ridiculus mus.

Sponsored  
It might be time to visit

<https://www.cypress.io/>



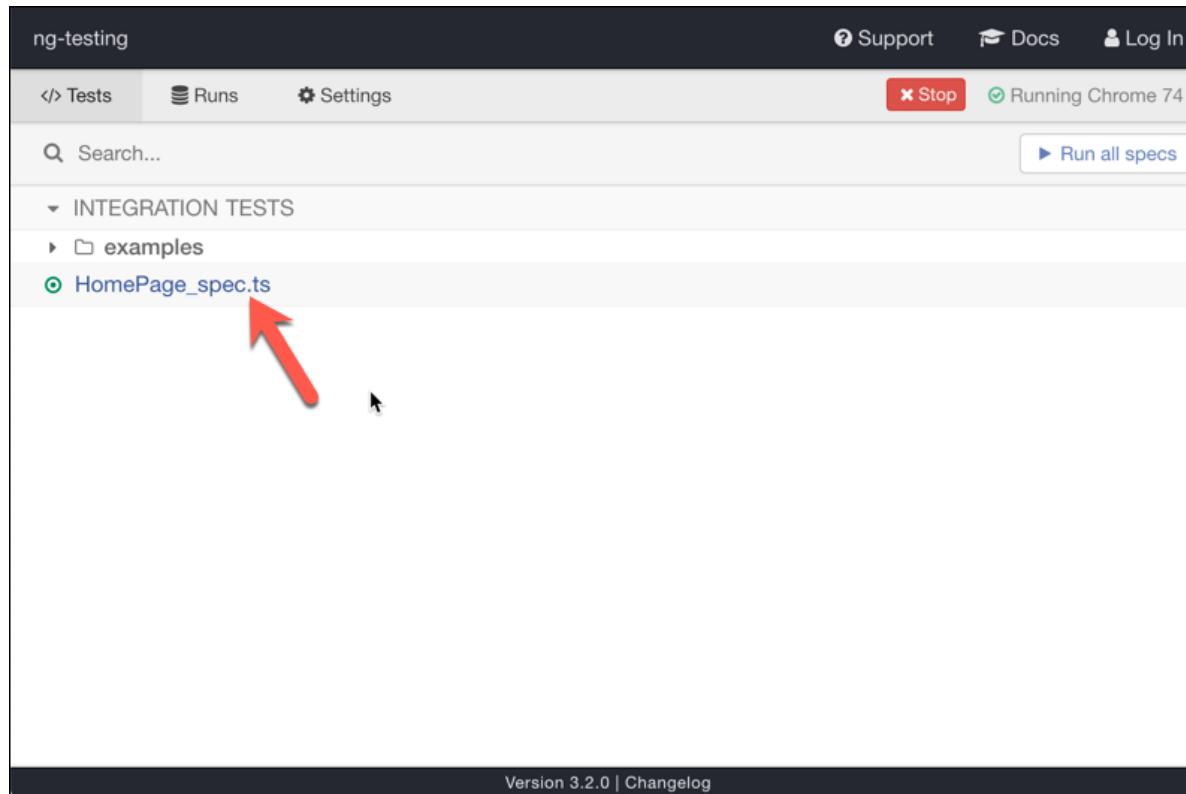
Testing Angular with Cypress.io | Joe Eames | AngularConnect 2018

Volgende

<https://www.youtube.com/watch?v=eZyD-8qglWY>

# In testing repo

- A small test for the homepage
  - Start project on localhost:4200
  - npm run cypress:open



Screenshot of a web browser showing Cypress test results and an Angular application.

The browser title is "ng-testing". The address bar shows "localhost:4200/\_/#/tests/integration/HomePage\_spec.ts". The test summary indicates 6 passed tests, 0 failed, and 0 pending. A red circle highlights the test summary area.

The test results are listed:

- Cypress testing the home page
  - ✓ should visit the home page
  - ✓ should have the title Welcome to app
- Testing the e2e component
  - ✓ should have the title Example e2e testing component
  - ✓ should have a default of 1 point
  - ✓ should add points when clicked 3 times
  - ✓ should reset when clicked

The Angular application displays the text "Welcome to app!!" and the Angular logo (a red hexagon with a white letter A).

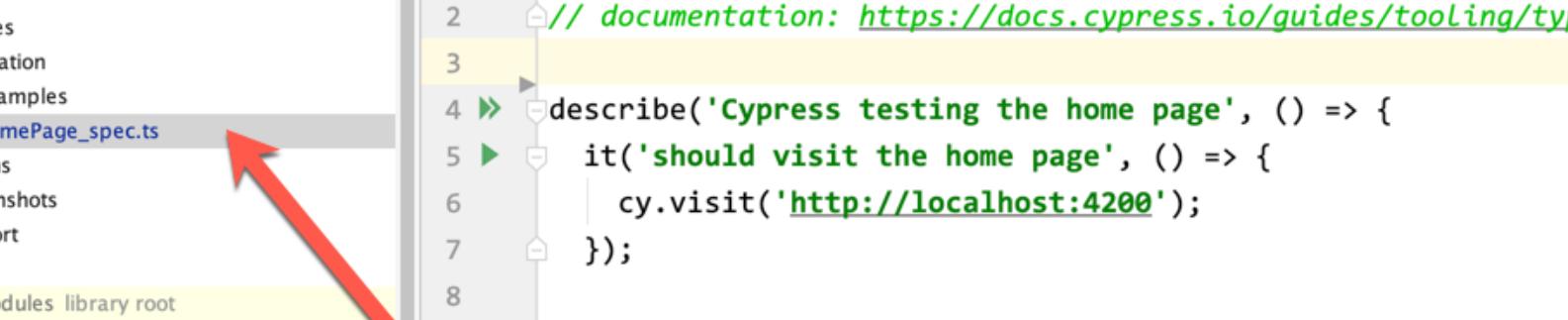
Text below the application:

Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

**Workshop 1**

# Check the code



The screenshot shows the Visual Studio Code interface with the following details:

- Project Tree:** On the left, under the "ng-testing" project, the "cypress" folder contains "fixtures", "integration" (which has "examples" and "HomePage\_spec.ts"), "plugins", "screenshots", "support", and "e2e".
- File List:** The "cypress" folder also lists ".angulardoc.json", ".editorconfig", ".gitignore", "angular.json", "cypress.json", "karma.conf.js", "package.json", "package-lock.json", and "protractor.conf.js".
- Code Editor:** The main pane displays the content of "HomePage\_spec.ts".

```
// HomePage_spec.ts - some cypress tests for our homepage
// documentation: https://docs.cypress.io/guides/tooling/typescript-s

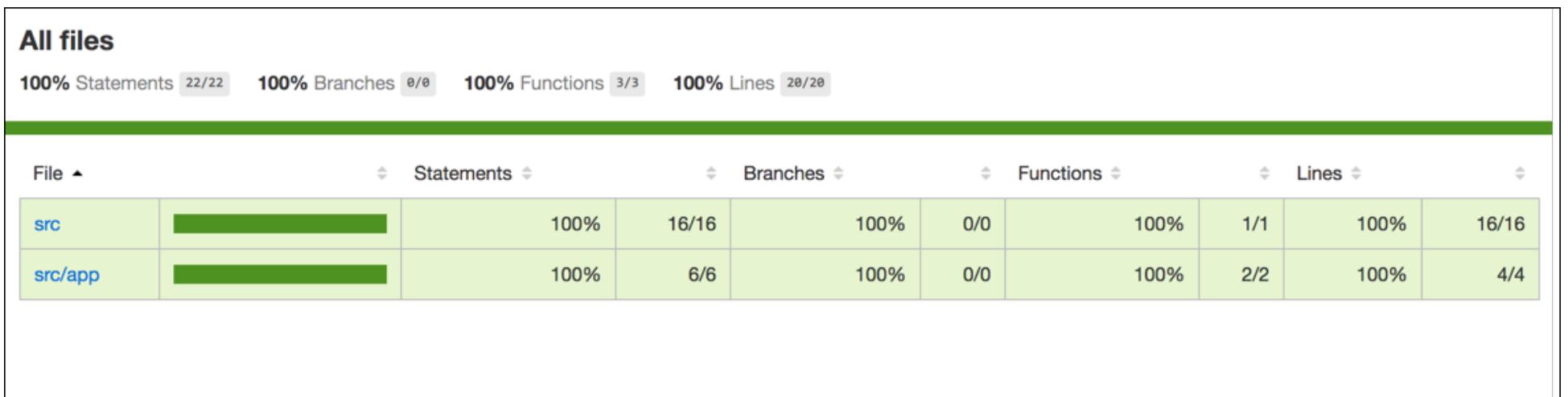
describe('Cypress testing the home page', () => {
  it('should visit the home page', () => {
    cy.visit('http://localhost:4200');
  });

  it('should have the title Welcome to app', () => {
    cy.visit('http://localhost:4200');
    cy.get('h1').contains('Welcome to app')
  });

  describe('Testing the e2e component', () => {
    it('should have the title Example e2e testing component', () => {
      cy.visit('http://localhost:4200');
      cy.get('h1').contains('Example e2e testing component')
    });
  });
});
```

# Code Coverage

- “How much of my code is covered by tests?”
- 100% would be ideal, but many teams are OK with 80-90%.
  - To be discussed
- Built in in CLI as a flag:
  - `ng test --code-coverage`





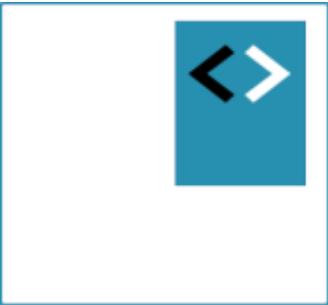
# Summary

What have we learned

# Summary

- We learned about:
  - ComponentFixture
  - .compileComponents()
  - .detectChanges()
  - .componentInstance
  - .debugElement
  - .nativeElement
  - By (helper class)
  - NO\_ERROR\_SCHEMA
  - Mocking strategies
  - End to End testing with Cypress or Protractor





# More info

Elsewhere on the interwebz...

# Testing documentation

The screenshot shows the Angular documentation website for the 'Testing' guide. The top navigation bar includes links for ANGULAR, FEATURES, DOCS, RESOURCES, EVENTS, and BLOG, along with a search bar. The main content area has a heading 'Testing'. Below it, a paragraph explains that the guide offers tips and techniques for testing Angular applications, focusing on those written with Angular. A section titled 'Live examples' describes a sample application with its tests available as live examples. A list of links provides access to various test specifications. To the right, a sidebar is highlighted with a red border, containing a tree view of the 'Testing' documentation structure.

Testing

This guide offers tips and techniques for testing Angular applications. Though this page includes some general testing principles and techniques, the focus is on testing applications written with Angular.

## Live examples

This guide presents tests of a sample application that is much like the [Tour of Heroes tutorial](#). The sample application and all tests in this guide are available as live examples for inspection, experiment, and download:

- [A spec to verify the test environment / download example](#).
- [The first component spec with inline template / download example](#).
- [A component spec with external template / download example](#).
- [The QuickStart seed's AppComponent spec / download example](#).
- [The sample application to be tested / download example](#).
- [All specs that test the sample application / download example](#).
- [A grab bag of additional specs / download example](#).

[Back to top](#)

## Introduction to Angular Testing

This page guides you through writing tests to explore and confirm the behavior of the application. Testing does the following:

1. Guards against changes that break existing code ("regressions").
2. Clarifies what the code does both when used as intended and when faced with deviant conditions.
3. Reveals mistakes in design and implementation. Tests shine a harsh light on the code from many angles. When a part of the application seems hard to test, the root cause is often a design flaw, something to cure now rather than later when it becomes expensive to fix.

● Testing

- Live examples
- Introduction to Angular Testing
- Tools and technologies
- Setup
- Isolated unit tests vs. the Angular testing utilities
- The first karma test
- Run with karma
- Test debugging
- Try the live example
- Test a component
- TestBed*
- createComponent*
- ComponentFixture, DebugElement, and query(By.css)*
- The tests
- detectChanges: Angular change detection within a test*
- Try the live example
- Automatic change detection
- Test a component with an external template

<https://angular.io/guide/testing>

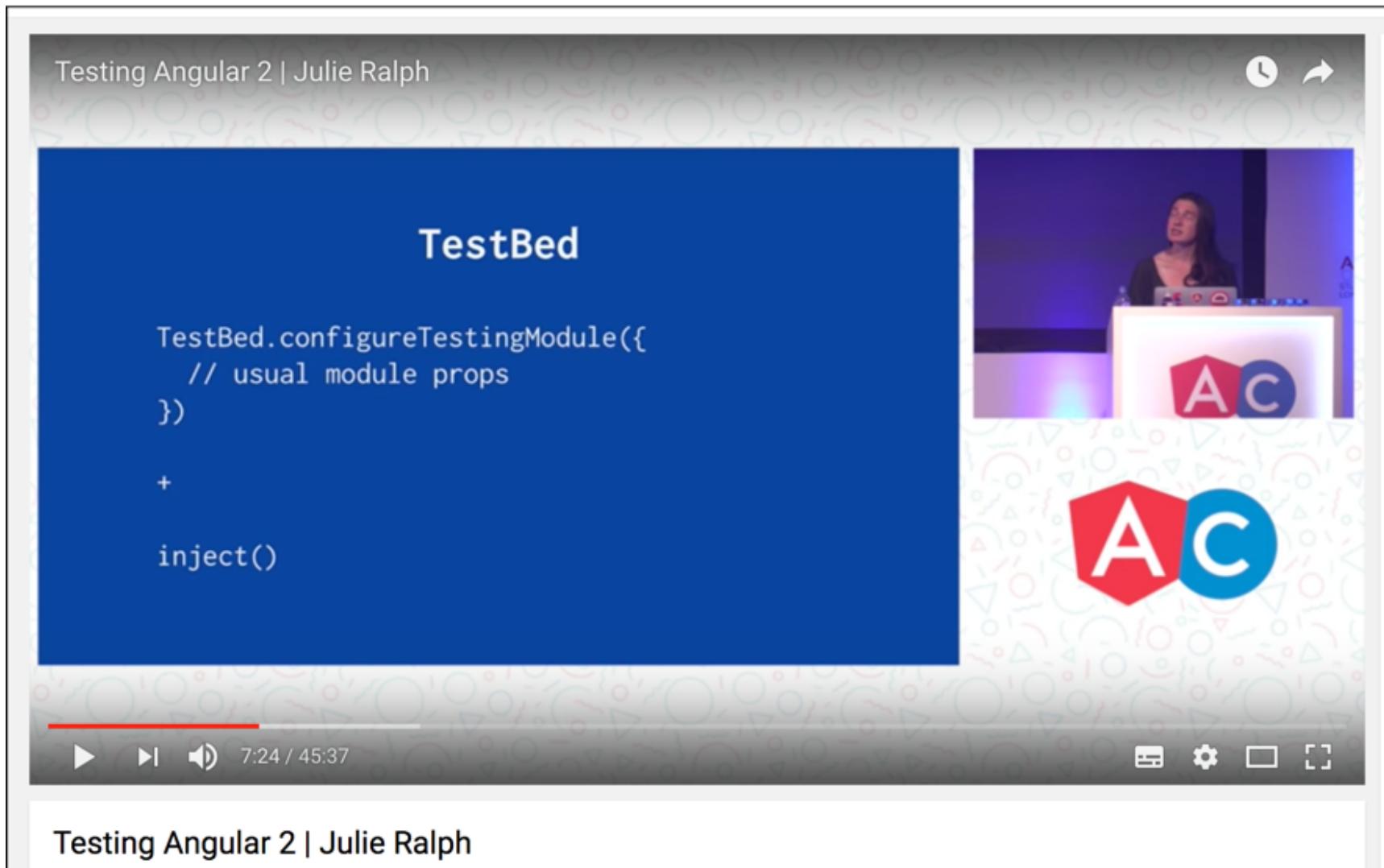
# Repo – Angular Testing recipes

The screenshot shows the GitHub repository page for 'juristr/angular-testing-recipes'. The repository has 3 stars, 18 forks, and 1 contributor. It contains 43 commits, 1 branch, and 0 releases. The latest commit was made a month ago. The repository is public and can be cloned or downloaded.

Commit	Message	Date
e2e	chore: base setup with CLI	a month ago
src	fix: missing parentheses	a month ago
.gitignore	chore: add .vscode to gitignore	a month ago
.travis.yml	chore: add travis config file	a month ago
README.md	docs: adjust intro	a month ago
angular-cli.json	chore: base setup with CLI	a month ago
karma.conf.js	feat: add karma mocha reporter	a month ago
package.json	chore(packages): upgrade TypeScript reference	a month ago
protractor.conf.js	chore: base setup with CLI	a month ago

<https://github.com/juristr/angular-testing-recipes>

# Videos on testing



<https://www.youtube.com/watch?v=f493Xf0F2yU>

# Good introductory article + video

The screenshot shows a DZone article page. At the top, the DZone logo and 'Web Dev Zone' are visible, along with a navigation bar for 'REFCARDZ', 'GUIDES', 'ZONES', and various technology categories like 'AGILE', 'BIG DATA', 'CLOUD', etc. A search icon and a user profile icon are also present. The main title of the article is 'Testing With Angular 2: Some Recipes (Talk and Slides)' by Juri Strumpflohner. Below the title is a brief summary: 'Juri Stumpflohner reflects on his recent talk about diving deeper into testing Angular 2 apps. He also links to a dedicated code repository on GitHub with the purpose of collecting testing recipes for various scenarios one might encounter while testing Angular applications.' The author's profile picture and name are shown, along with the publication date ('Jan. 16, 17') and category ('Web Dev Zone'). Below the summary are social sharing buttons for 'Like (-2)', 'Comment (0)', 'Save', and 'Tweet', and a view count of '4,327 Views'. A call-to-action button says 'JOIN FOR FREE'. At the bottom, there's a note about Qlik and a 'Subscribe' button.

**Testing With Angular 2: Some Recipes (Talk and Slides)**

Juri Stumpflohner reflects on his recent talk about diving deeper into testing Angular 2 apps. He also links to a dedicated code repository on GitHub with the purpose of collecting testing recipes for various scenarios one might encounter while testing Angular applications.

by Juri Strumpflohner MVB · Jan. 16, 17 · Web Dev Zone

Like (-2) Comment (0) Save Tweet 4,327 Views

Join the DZone community and get the full member experience. JOIN FOR FREE

Start coding today to experience the powerful engine that drives data application's development, brought to you in partnership with [Qlik](#).

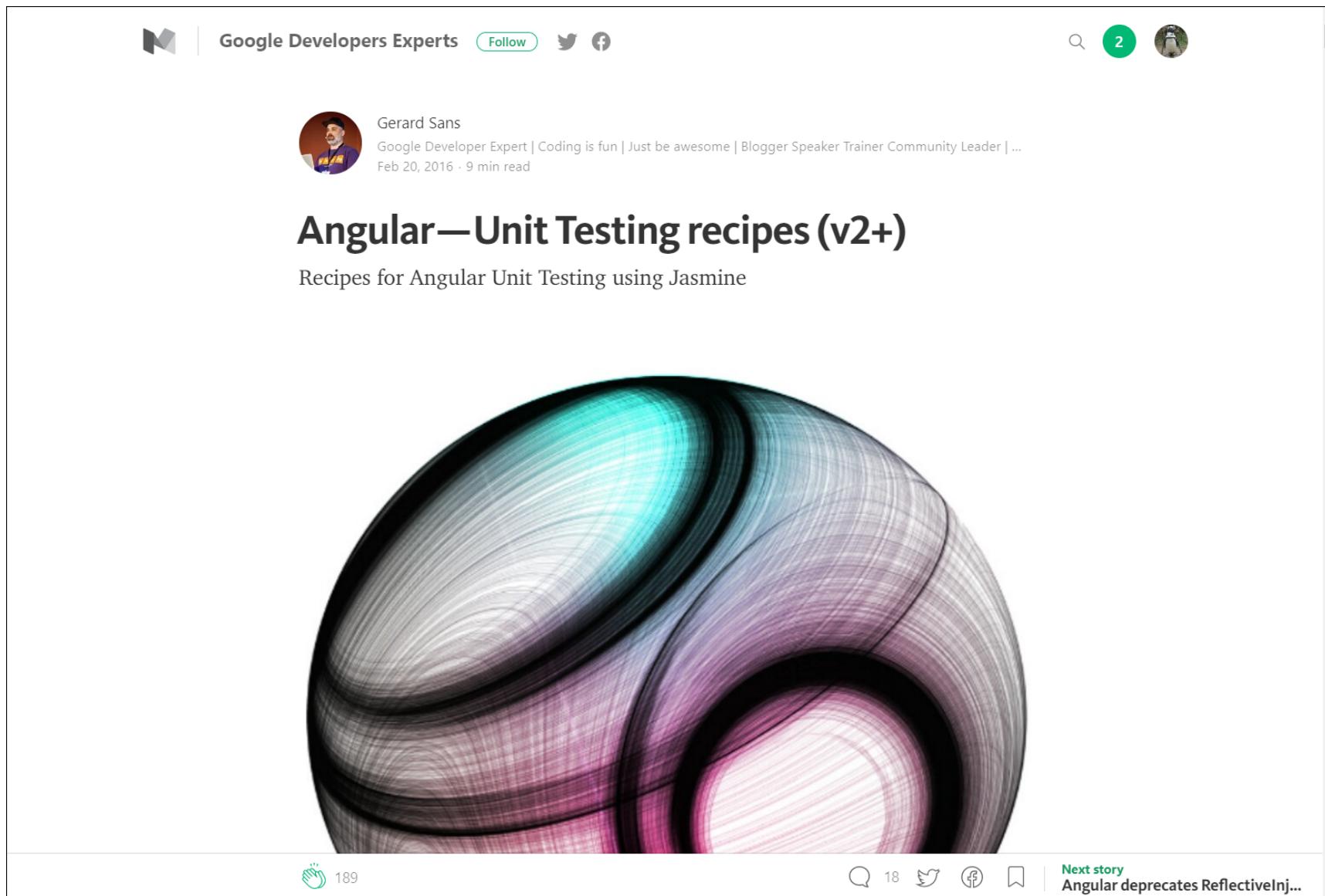
I recently wanted to dive deeper into testing Angular applications, in specific on how to write proper unit tests for some common scenarios you might encounter.

Dave, the organizer of [the Angular Hamburg Meetup group](#), asked me whether I'd be interested in

Subscribe ^

<https://dzone.com/articles/talk-testing-with-angular-some-recipes>

# Gerard Sans on testing



The image shows a screenshot of a Medium article. At the top left is the Medium logo (a stylized 'M'). Next to it is the author's name, "Google Developers Experts", followed by a "Follow" button, a Twitter icon, and a Facebook icon. On the right side of the header are a search icon, a green circular notification badge with the number "2", and a user profile icon. Below the header, the author's profile picture is shown, followed by the author's name, "Gerard Sans", and their bio: "Google Developer Expert | Coding is fun | Just be awesome | Blogger Speaker Trainer Community Leader | ...". It also shows the publication date, "Feb 20, 2016 · 9 min read". The main title of the article is "Angular—Unit Testing recipes (v2+)" in a large, bold, dark font. Below the title is a subtitle, "Recipes for Angular Unit Testing using Jasmine". The main content area features a large, abstract, colorful image of concentric, swirling lines in shades of black, white, teal, and purple. At the bottom of the article, there is a footer bar with several icons: a clapping hands icon (189), a comment icon (18), a Twitter icon, a share icon, a bookmark icon, and a link to the next story, "Angular deprecates ReflectiveInj...".

<https://medium.com/google-developer-experts/angular-2-unit-testing-with-jasmine-defe20421584>

# Testing Routing

The screenshot shows a website layout for 'CODE CRAFT'. On the left, there's a sidebar with navigation links for various Angular topics like Quickstart, ES6, Components, and Routing. Under 'Unit Testing', there's a sub-menu with 'Overview' and several other items. At the top right, there's a navigation bar with links for HOME, BLOG, COURSES (with a dropdown arrow), and ABOUT. The main content area has a breadcrumb trail: 'Angular 4 / Unit Testing / Testing Routing'. The main title 'Testing Routing' is displayed prominently. Below it is a large image of a video player interface with the text 'Tired of reading? Watch the videos instead'. A red button labeled 'Watch NOW!' is visible. To the right of the main content, there's a vertical column of social sharing icons for Google+, Facebook, Twitter, and LinkedIn. At the bottom, there are navigation arrows for 'TESTING HTTP' and 'WRAPPING UP'.

- Quickstart
- ES6 JavaScript & TypeScript
- Angular CLI
- Components
- Built-in Directives
- Custom Directives
- Reactive Programming with RxJS
- Pipes
- Forms
- Dependency Injection & Providers
- HTTP
- Routing
- Unit Testing**
  - Overview
  - Jasmine & Karma
  - Testing Classes & Pipes
  - Testing with Mocks & Spies
  - Angular Test Bed
  - Testing Change Detection
  - Testing Asynchronous Code
  - Testing Dependency Injection

Angular 4 / Unit Testing / Testing Routing

# Testing Routing

Tired of reading? Watch the videos instead

Click to find out more info and purchase the associated video course.

Watch NOW!

TESTING HTTP

WRAPPING UP

<https://codecraft.tv/courses/angular/unit-testing/routing/>