# Lab Module 1

## Introduction

## The Angular CLI comes with a set of templates to scaffold standard Angular applications.

## It will also download and install everything you need to test an Angular application with the [Jasmine testing framework](https://jasmine.github.io/) and [Karma test runner](https://karma-runner.github.io/latest/index.html).

## Pre-requisites

Latest version of [Nodejs](https://nodejs.org/en/) / [Npm](https://www.npmjs.com/get-npm), [Visual Studio Code](https://code.visualstudio.com/), basic knowledge of [TypeScript](https://www.typescriptlang.org/) and a GitHub account.

## Objectives

After completing this lab, you will be able to:

* Have a basic understanding of Jasmine testing framework and mocking techniques
* Write isolated unit test for an Angular component
* Have a good understanding of the 2 types of integration tests

Scenario

We are going to work with a modified version of Tour of Heroes, Angular’s tutorial app which has many of the features you would expect to find in data-driven Single Page Application (SPA). The SPA acquires, searches and displays a list of heroes, edits selected hero's details, and navigates among different views of heroic data.

## Setting up

If you haven’t installed yet, please download current’s version of Nodejs (12.8.1) for windows (or Mac, or Ubuntu) at <https://nodejs.org/dist/v12.8.1/node-v12.8.1-x64.msi>

Then download, clone the repository and install the application’s dependencies with npm.

git clone <https://github.com/Gwayaboy/AngularTestingWorkshop.git> ngTestLab

cd ngTestLab/TourOfHeroesApp/

npm i

Once this is done, you can verify the application works as expected by running

npm start

Once the application has started, you can navigate to <http://localhost:4200/> to have a chance to explore the application.

Since we are not going to write any end to end UI tests, we can now stop it by running

npm stop

We can now open Visual studio from the command line

code .

### Exercise 1: Writing and Running Isolated Unit Tests

#### Introduction

In unit testing it is common to use objects that look and behave like their production equivalents but are simplified. This reduces complexity, allows to verify code independently from the rest of the system and sometimes it is even necessary to execute self-validating tests at all.

A Test Double is a generic term used for these objects.

We are going to explore the 3 different types of unit tests (value, stated & interaction based) as well as what are the different types of test doubles before using a Jasmine’s spy objects.

#### Task 1.1: Value-based unit testing Angular strength pipe

Value-based unit test verifies that the system under test (SUT) returns the expected results.

In our case that the transform method returns the value with a some sort of strength indicator

1. Open from your VS Explorer, the *NGTESTLAB/src/app/strength/strength.pipe.ts* file

* *Notice the @Pipe annotation to register and name the pipe "strength"*
* *The StrengthPipe inherits from a base PipeTransform class*

*All of these are extensibility point as part of the Angular framework to craft custom pipes.*

We are going to ignore all this plumbing and focus on verifying that the transform method returns the value concatenated with the correct strength indicator

transform(value: number): string {

if(value < 10) {

return value + " (weak)";

} else if(value >= 10 && value < 20) {

return value + " (strong)";

} else {

return value + " (unbelievable)";

}

}

1. Let’s add a *strength.pipe.spec.ts* file alongside the strength.pipe.ts as it is Karma’s convention to pick up spec files as automated test to run.

It is also a good practice in front-end web development (both React and Agular) to have the spec file at the same level as the system under test.

1. Let’s write our value-based unit test using [Jasmine’s Behaviour-Driven Development](https://jasmine.github.io/index.html) (BDD) structure.

The test file follows the AAA (Arrange-Act-Assert) using the keywords [**describe**](https://jasmine.github.io/api/edge/global.html#describe), [**it**](https://jasmine.github.io/api/edge/global.html#it) and [**expect**](https://jasmine.github.io/api/edge/global.html#expect).

These are nothing more than JavaScript functions accepting parameters and nesting each other in the enumerated order above

describe('StrengthPipe', () => {

it('should display (10 strong) when strength is 10 ', () => {

// Arrange

// Act

// Assert

});

1. in our Arrange section we create an instance of our StrengthPipe class

const pipe = new StrengthPipe();

1. in our Act section we invoke the transform method

const result = pipe.transform(10);

1. our assertion verifies the returned result is *“10 strong”* as expected

expect(result).toEqual('10 (strong)');

#### Tasks 1.2: Running our first value-based unit test

We’re using Karma as our test runner. There are a couple of ways you can run your unit tests

* using the terminal
* using an VS Code extensions like [Angular/Karma Test Explorer](https://marketplace.visualstudio.com/items?itemName=raagh.angular-karma-test-explorer) or [Wallaby](https://marketplace.visualstudio.com/items?itemName=WallabyJs.wallaby-vscode) to run the tests on-demand or continuously.

In either case, the *package.conf* and *karma.conf* files defines how your test are going to be run. The *package.conf* has a “scripts” property which describes the command to run tests:

"scripts": {

"ng": "ng",

"build": "ng build --prod",

"start": "ng serve",

"test": "ng test",

"lint": "tslint ./src/\*\*/\*.ts -t verbose",

"e2e": "ng e2e"

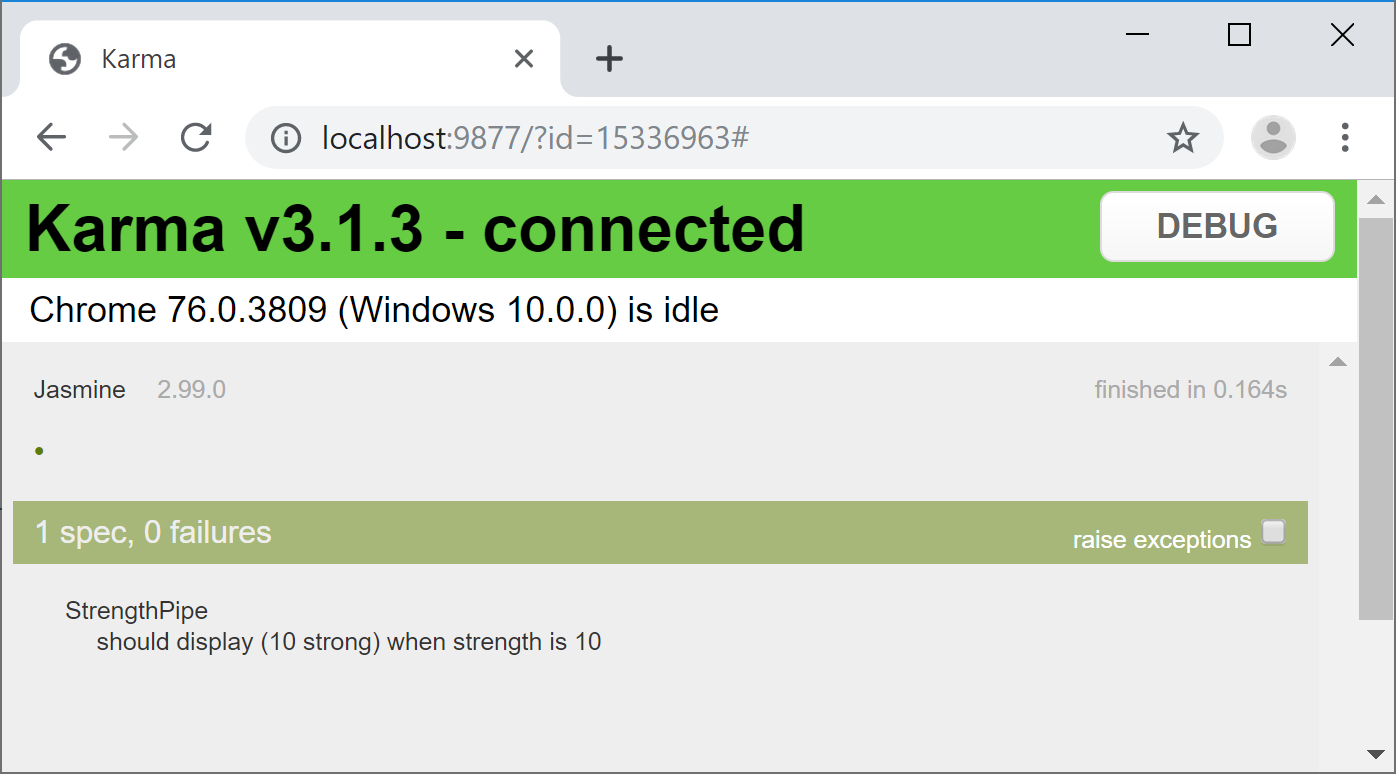
},

1. Open a new terminal in VS Code and execute the test running

npm test

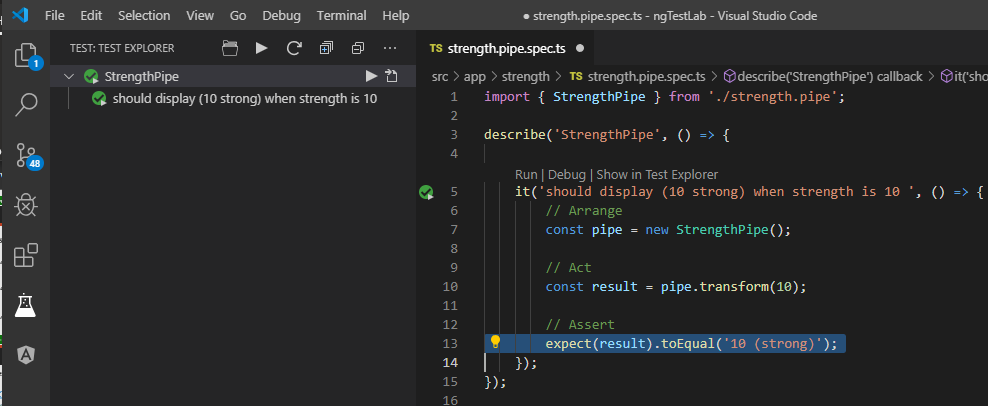
This will execute Angular cli ng test command which will compile and execute the test in Karma opening your default browser.

A web socket connection is maintained and listens for any spec file change to re-run and update the browser test output.



1. Install Angular/Karma Test Explorer Visual Studio Code Extension to have a similar experience as Visual Studio’s Text Explorer left Pane.

This extension also allows continuous testing and super-impose any expectation failure



#### Tasks 1.3: Parameterising value-based tests

Writing unit tests is all about covering all execution path. We need to other tests to verify all outcome. It will be easily to duplicate the test we wrote previously for strength value less than 10 or more than 19 but we could take advantage of the language feature to parameterise the test to verify the 3 scenarios

1. Create an array of parameters that will expose 2 properties, the input strength and the expected result

const testCases = [

{strength: 1, expected: '1 (weak)'},

{strength: 10, expected: '10 (strong)'},

{strength: 100, expected '100 (unbelievable)'},

];

1. Wrap up the it function using ES6 foreach to iterate through the parameters using strength and expected result to parameterise the test.

testCases.forEach(({strength, expected}) => {

it(`should display ${expected} when strength is ${strength}`,() => {

// Arrange

const pipe = new StrengthPipe();

// Act

const result = pipe.transform(strength);

// Assert

expect(result).toEqual(expectedResult);

});

#### Tasks 1.3: State-based unit testing with Angular HeroesComponent

#### Task 1.4: Interaction-based unit testing with Angular HeroesComponent and Jasmine’s Spy Object

### Exercise 2: Writing Shallow Integration Tests

#### Introduction

Google officially call shallow integration tests, Angular Integration tests which consist of testing a component works well with both its presentational and container parts.

For example, a shallow test will verify that the component’s template renders the correct output from the service feeding its data.

As a shallow integration test, the boundaries are limited to the component itself as whole and will substitute any injected dependencies (service and children components) with a mocked implementation.

#### Task 2.1: Validate Hero component’s template renders correctly

#### Task 2.2: Mocking Hero child components.

### Exercise 3: Writing Deep Integration Tests

#### Introduction

Deep integration tests verify that multiple components work together and usually limits their boundaries to injected services.

#### Task 3.1: Verifying that Hero children components are initialised correctly from the parent Heroes component

#### Task 3.2: Deep integration testing of Hero Service.