|  |
| --- |
|  |
| Angular 8 |
|  |
| Displaying Data, Karma |

**TechBrain Express**

December 5, 2019

Authored by: Ashu

Angular –Display Data-Karma

* **Demonstrate creation of angular application**
* **Demonstrate displaying data using interpolation, ngIf and ngFor**
* **Demonstrate using pipes to format number and date**
* **Demonstrate karma test case creation with DOM reading**

**Demonstrate displaying data using interpolation**

Refer: <https://angular.io/guide/displaying-data>

**Interpolation:** Interpolation allows you to incorporate calculated strings into the text between HTML element tags and within attribute assignments. Template expressions are what you use to calculate those strings.

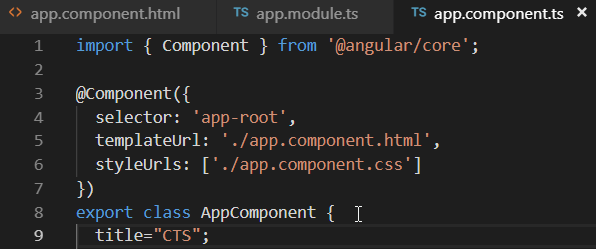
Whenever you need to communicate properties (variables, objects, arrays, etc..) from the component class to the template, you can use interpolation.

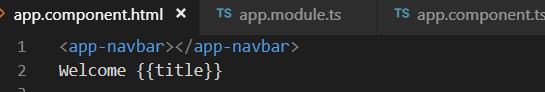
The format for defining interpolation in a template is: **{{ propertyName }}**

Binding data from class in template .

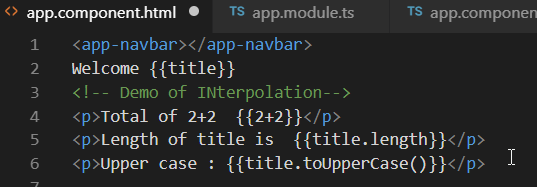
Value of customer name must be changed anytime.

{{}} is String INterpolation

****

****

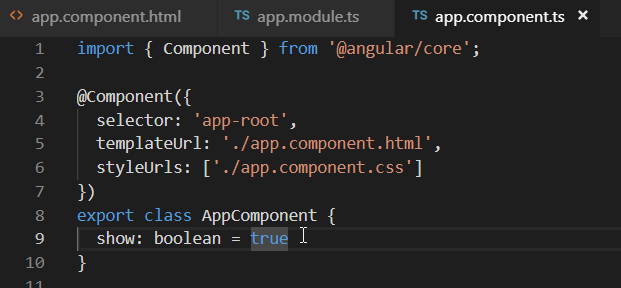
### Lab: Display length of title and convert it in upper case along with demo of 2+2

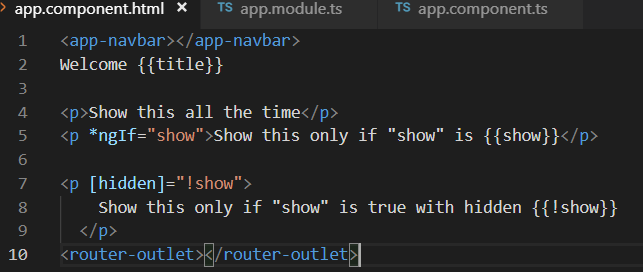


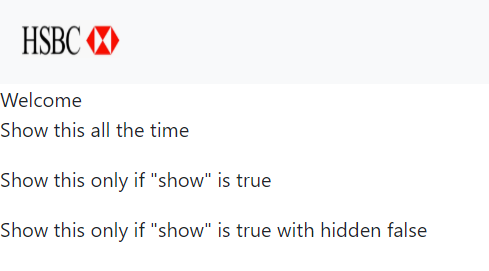
**Demostrate ng if**

That means, that it can be added to any tag in your template.

<p \*[ngIf](https://angular.io/api/common/NgIf)="heroes.length > 3">There are many heroes!</p>







NgIf is a directive.

That means, that it can be added to any tag in your template.

While the hidden attribute is literally hiding the selected part of the DOM, just like the CSS "display: none" property, the element still sit on the DOM. They are just invisible.

Angulars' ngIf directive, on the other hand, is completely removing the selected part from the DOM. The great advantage of that is, that this method is not interfering with any CSS-Style-sheets at all. It is simply removing anything.

## Null Check with ngIf

Displaying a box with the users' information wouldn't make much sense if there is no information to display. Normally this would result in an empty box to be shown.

**The ngIf directive can be very handy to prevent "Cannot read property of undefined" errors, as well.**

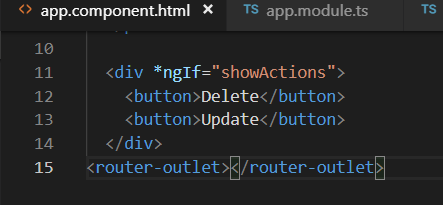
Check the user-property for null before using its properties will prevent this.

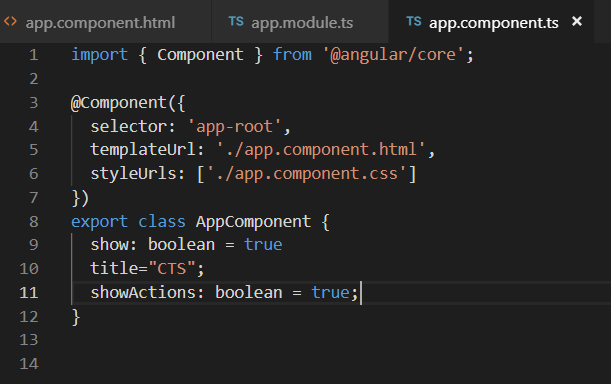
<p \*ngIf="user">

{{user.name}}

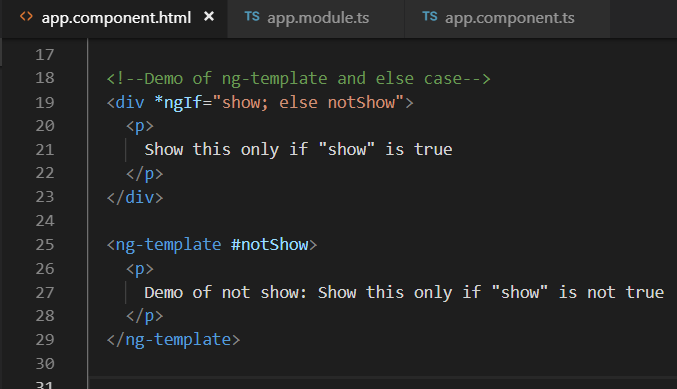
</p>

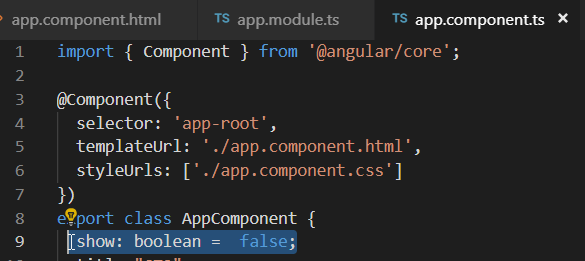
### Lab: Display buttons if showActions =true





### Lab: Demo of else and ng-template



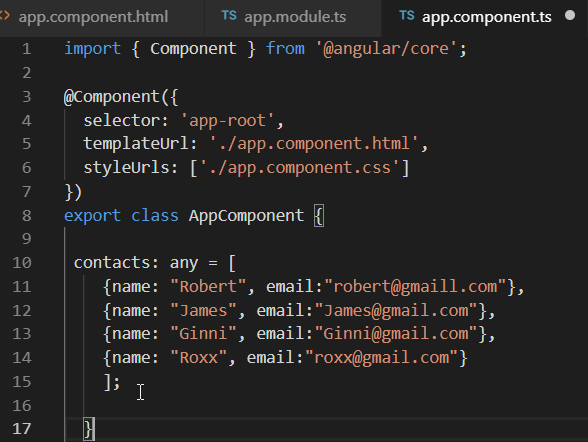


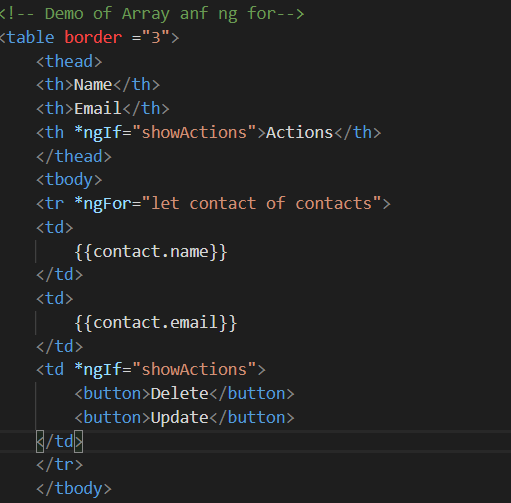
This block is shown if the statement defined in the main block happens to be false. the else block has to be an ng-template. The ng-template is a special element that is interpreted by angular and has no DOM counterpart. That means, that this tag is not included in the final HTML result, but only its content.

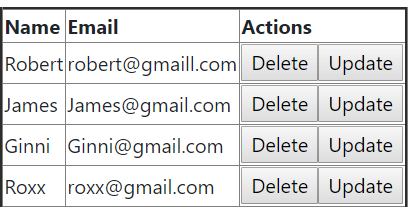
We have to use ng-content, because a normal div would be picked up by the browser and be rendered all the time.

Also, we have to give that block an ID (#notShow) so we can reference it in our ngIf directive as the "else"-block.

# Demonstrate ngFor







**Demonstrate using pipes to format number and date**

**Pipe:**

Every application starts out with what seems like a simple task: get data, transform them, and show them to users.

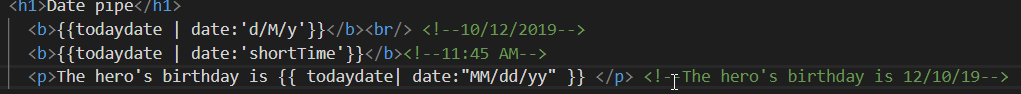
* A pipe is a class decorated with pipe metadata.
* The pipe class implements the [PipeTransform](https://angular.io/api/core/PipeTransform) interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.
* There will be one additional argument to the transform method for each parameter passed to the pipe. Your pipe has one such parameter: the exponent.
* To tell Angular that this is a pipe, you apply the @[Pipe](https://angular.io/api/core/Pipe) decorator, which you import from the core Angular library.
* The @[Pipe](https://angular.io/api/core/Pipe) decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier. Your pipe's name is exponentialStrength.

### [CurrencyPipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_currencypipe)

{{ 1234.56 | currency:'CAD' }}

### 

### [DatePipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_datepipe)

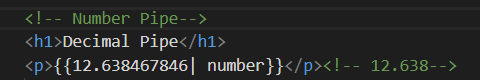


### [DecimalPipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_decimalpipe)

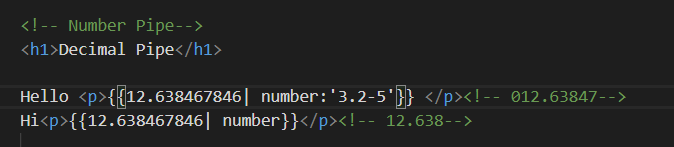
**number\_expression | number[:digitInfo]**  
  
Finally we get a decimal number as text. Find the description.  
**number\_expression**: An angular expression that will give output a number.  
**number**: A pipe keyword that is used with pipe operator.  
**digitInfo**: It defines number format.  
  
Now we will understand how to use **digitInfo**. The syntax for **digitInfo** is as follows.  
  
**{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}**  
  
Find the description.  
**minIntegerDigits**: Minimum number of integer digits. Default is 1.  
**minFractionDigits**: Minimum number of fraction digits. Default is 0.  
**maxFractionDigits**: Maximum number of fraction digits. Default is 3.

1. Using default format:

minIntegerDigits = 1  
minFractionDigits = 0  
maxFractionDigits = 3



1. Format **'3.2-5'**  
     
   minIntegerDigits = 3  
   minFractionDigits = 2  
   maxFractionDigits = 5  
     
   Now find the number that will be formatted



### [JsonPipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_jsonpipe)

**App.componenet.ts**

jsonval = {name:'Rox', age:'25', address:{a1:'Mumbai', a2:'Karnataka'}};

<!-- jsonPipe-->

<p>{{jsonval|json}}</p>

{ "name": "Rox", "age": "25", "address": { "a1": "Mumbai", "a2": "Karnataka" } }

### [LowerCasePipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_lowercasepipe)

<h1>Lowercase Pipe</h1>

<b>{{title | lowercase}}</b>

### [UpperCasePipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_uppercasepipe)

<b>{{title | uppercase}}</b><br/>

### [PercentPipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_percentpipe)

<h1>Percent Pipe</h1>

<b>{{00.54565 | percent}}</b> //54.56%

### [SlicePipe](https://codecraft.tv/courses/angular/pipes/built-in-pipes/#_slicepipe)

months = ["Jan", "Feb", "Mar", "April", "May", "Jun",

"July", "Aug", "Sept", "Oct", "Nov", "Dec"];

<b>{{months | slice:2:6}}</b> <!--Mar,April,May,Jun-->

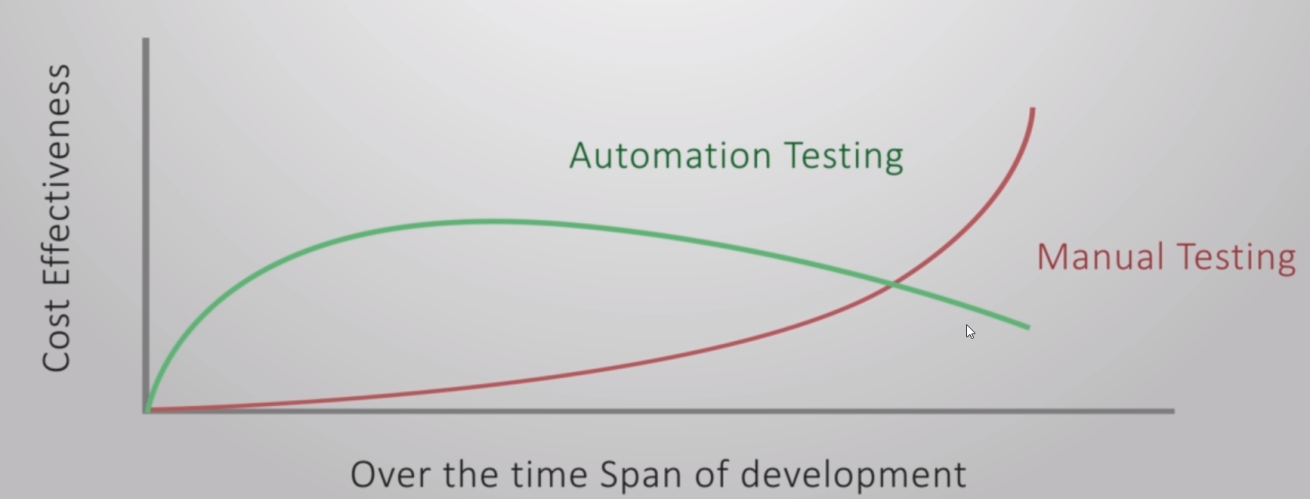
**Demonstrate karma test case creation with DOM reading**

Refer Project : F:\AngularOct19\angulardisplaydata

## Introduction to Automated Testing

<https://www.guru99.com/automation-testing.html>

|  |  |
| --- | --- |
| Manual Testing | Automated Testing |
| we have to launch the application in broswser, do a couple clicks and verify the application with change in data. May 50 sec | Writing test code and call this function with different input  Run this code in automated fashion. |
|  |  |



Argument –As a developer we have to write a production code as well as test code so testing will take more time rather than without test code.

Supportive- With large application, team does not know the expectedfunctionality as team may leave over a period. Time with authomation would be lesser. It helps you catch defects before releasing your software.

## Different types of testing

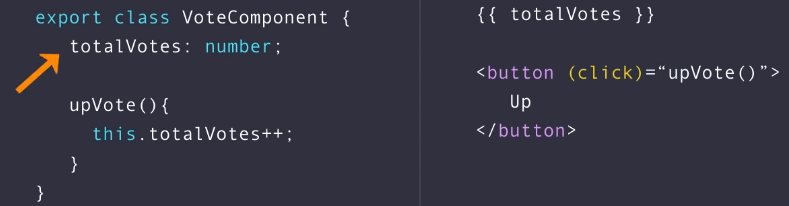
Unit Testing:



We test a component in isolation, without a external resource( eg: file syste, , db, api endpoints)

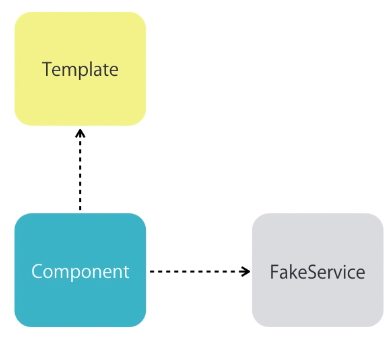
In angular, without template test component. If your compoennet a using service and routing. Test it with fake service ir fake router,

They are easier, faster and don’t give us much confidence. In given example we can test angular code without template by calling upVote() method using button click.



### Integration Testing:

We test component with external resources (filesystem, database, api endpoints)



### End to End Testing

More confidence

Very slow

Very fragile

## How to write test cases?

Clean Coding Practices:

Small functiona/methods(10 lines of code or less)

Proper Naming

Single repsonsibility

## Jasmine

* Jasmine is a javascript testing framework that supports a software development practice called [Behaviour Driven Development](https://en.wikipedia.org/wiki/Behavior-driven_development), or BDD.
* Jasmine, and BDD in general, attempts to describe tests in a human readable format so that non-technical people can understand what is being tested.
* Jasmine provides a small syntax to test the smallest unit of the entire application instead of testing it as a whole.

|  |  |
| --- | --- |
| Code | Jasmine Test Code |
|  |  |

## Why Use Jasmine?

* It's fast and has low overhead and no external dependencies.
* It's a batteries included library and offers everything you need for testing your code.
* It's available both for Node and the browser.
* It can be used with other languages like Python and Ruby.
* It does not require the DOM.
* It provides a clean and easy to understand syntax and also a rich and straightforward API.
* We can use natural language to describe the tests and the expected results.

## Why of using JASMINE?

* In the old way by including both the Jasmine core and your test files using a <script> tag,
* As a CLI tool using Node.js,
* As a library in Node.js,
* As a part of a build system like Gulp.js or Grunt.js via [grunt-contrib-jasmine](https://github.com/gruntjs/grunt-contrib-jasmine) and [gulp-jasmine-browser](https://github.com/jasmine/gulp-jasmine-browser)

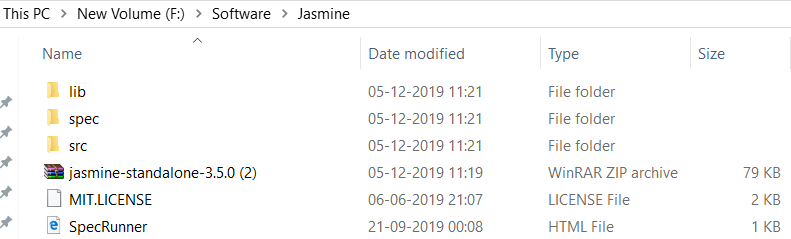
 By convention, the tests written in Jasmine are called specs.

Open app.component.apec.ts up and have a good look at the Jasmine tests in Angular. Even if the code doesn't make any sense, that is fine.

## Download Jasmine

## Using Standalone JASMINE

1. Download Jasmine <https://github.com/jasmine/jasmine/releases>
2. Click on [jasmine-standalone-3.5.0.zip](https://github.com/jasmine/jasmine/releases/download/v3.5.0/jasmine-standalone-3.5.0.zip)



1. /Src: Contains the source files that you want to test. This may be either deleted if your already have your project's folder setup or can also be used when appropriate for hosting your source code.

/Lib:  Contains the core Jasmine files

/Spec:  Contains the tests that you are going to write.

/SpecRunner.html: This file is used as a test runner. You run your specs by simply launching this file.

## Suites

A suite groups a set of specs or test cases. It's used to test a specific behavior of the JavaScript code that's usually encapsulated by an object/class or a function. It's created using the Jasmine global function describe() that takes two parameters, the title of the test suite and a function that implements the actual code of the test suite.

### Specs

A spec declares a test case that belongs to a test suite. This is done by calling the Jasmine global function it() which takes two parameters, the title of the spec (which describes the logic we want to test) and a function that implements the actual test case.

A spec may contain one or more expectations. Each expectation is simply an assertion that can return either true or false. For the spec to be passed, all expectations belonging to the spec have to be true otherwise the spec fails.

### Expectations

Expectations are created using the expect() function that takes a value called the **actual** (this can be values, expressions, variables, functions or objects etc.). Expectations compose the spec and are used along with matcher functions (via chaining) to define what the developer expect from a specific unit of code to perform.

A matcher function compares between an **actual** value (passed to the expect() function it's chained with) and an **expected** value (directly passed as a parameter to the matcher) and returns either **true** or **false** which either **passes** or **fails** the spec.

You can chain the expect() function with multiple matchers. To negate/invert the boolean result of any matcher, you can use the not keyword before calling the matcher.

### Built-In Matchers

Jasmine provides a rich set of built-in matchers. Let's see some of the important ones:

* toBe() for testing for identity,
* toBeNull() for testing for null,
* toBeUndefined()/toBeDefined() for testing for undefined/not undefined,
* toBeNaN() for testing for NaN (Not A Number)
* toEqual() for testing for equality,
* toBeFalsy()/toBeTruthy() for testing for falseness/truthfulness etc.

You can find the full list of matchers from the [docs](https://jasmine.github.io/api/edge/matchers.html).

Let's now implement our specs with some of these matchers when appropriate. First import the functions we are testing in our MyJSUtilitiesSpec.js file:

const utils = require("../index.js");

Next, start with the String Utils suite and change expect().nothing() with the appropriate expectations.

## Using Jasmine via The CLI

## Karma Test Runner

K arma is a direct product of the AngularJS team from struggling to test their own framework features with existing tools. Karma is a tool that lets you test your application on multiple browsers. Karma has plugins for browsers like Chrome, Firefox, Safari, and many others. But I prefer using a headless browser for testing. A headless browser lacks a GUI, and that way, you can keep the test results inside your terminal. In this tutorial, we will configure Karma to run with Chrome and, optionally, a headless version of Chrome.

## Angular Testing Utilities

Angular testing utilities provide you a library to create a test environment for your application. Classes such as TestBed and ComponentFixtures and helper functions such as async and fakeAsync are part of the @angular/core/testing package. Getting acquainted with these utilities is necessary if you want to write tests that reveal how your components interact with their own template, services, and other components.

## Protractor

## Exercise

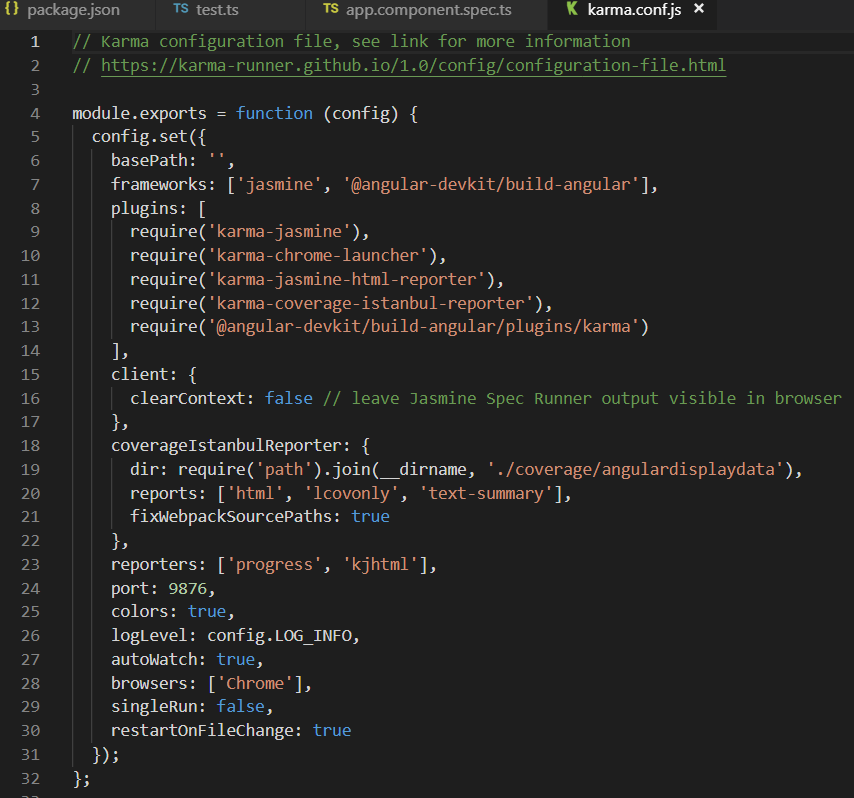
* 1. Create new Project



* 1. Open project in VSCODE
  2. Explore package.json where  .
  3. Look for src/test.ts file for 

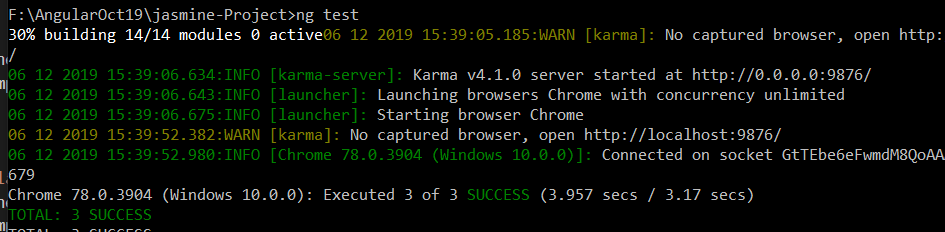
Means angular will look for spec.ts file for testing purpose

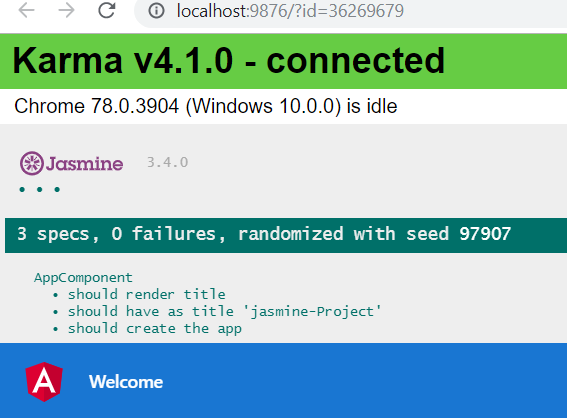
* 1. Open karma-config.json



Where test runner will run at 9876 port in chrome.

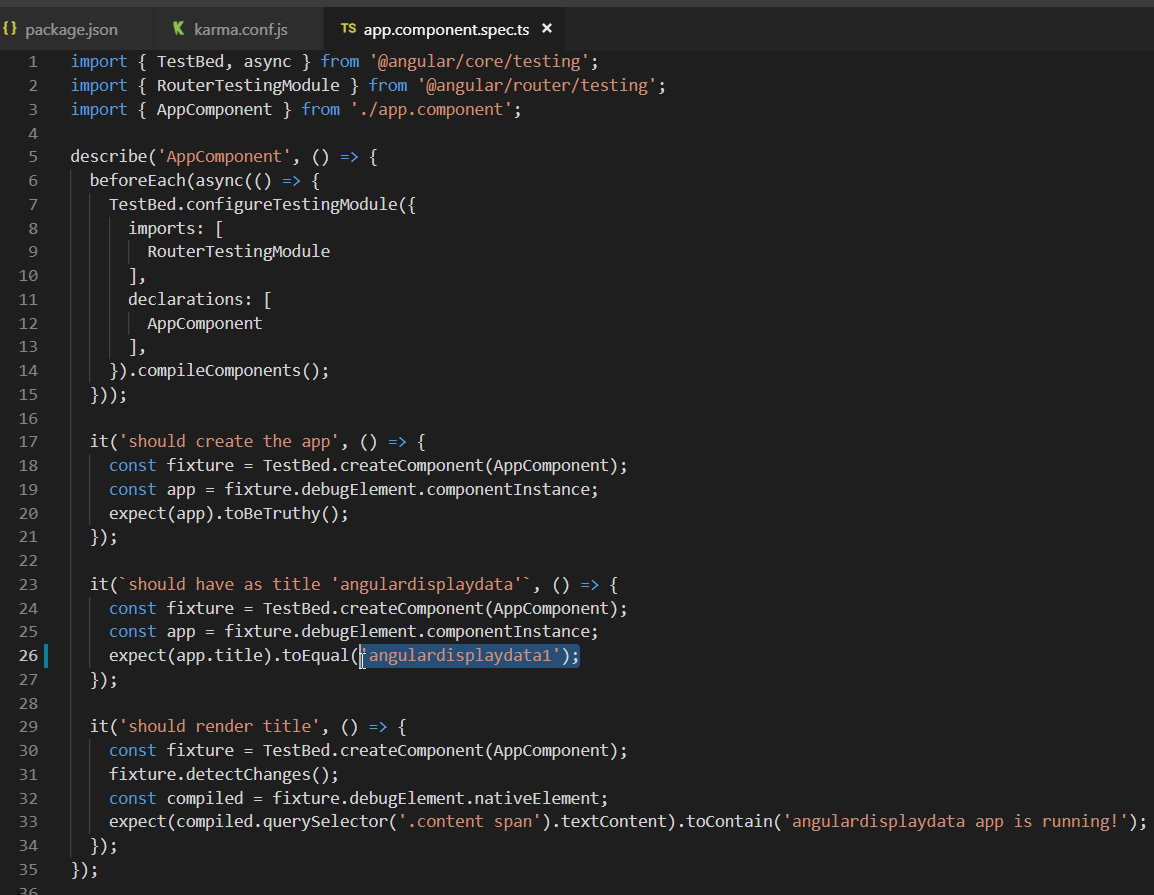
* 1. > ng test



* 1. Opens the browser 

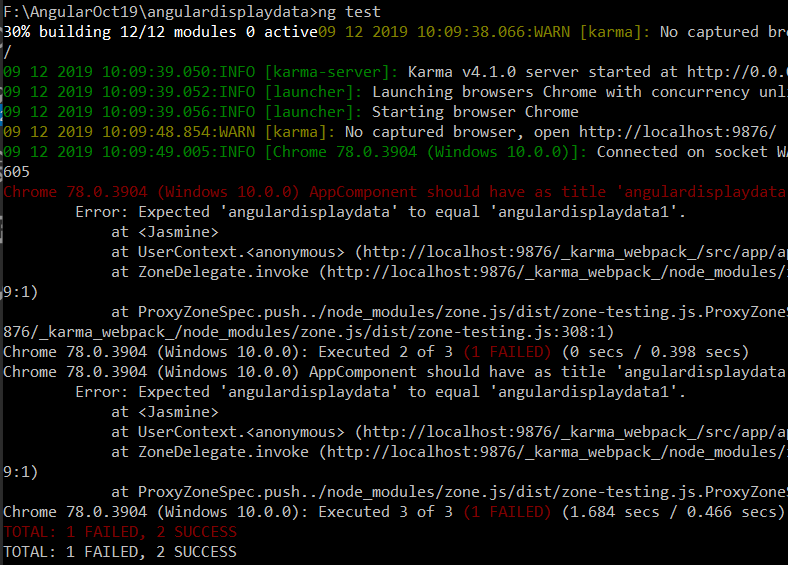
Here we can observe that 3 test cases are running however we have not specified.

* 1. Open app.spec.ts for review test cases.



* 1. In case you edit the expected value and retest using > ng test

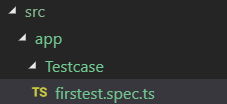


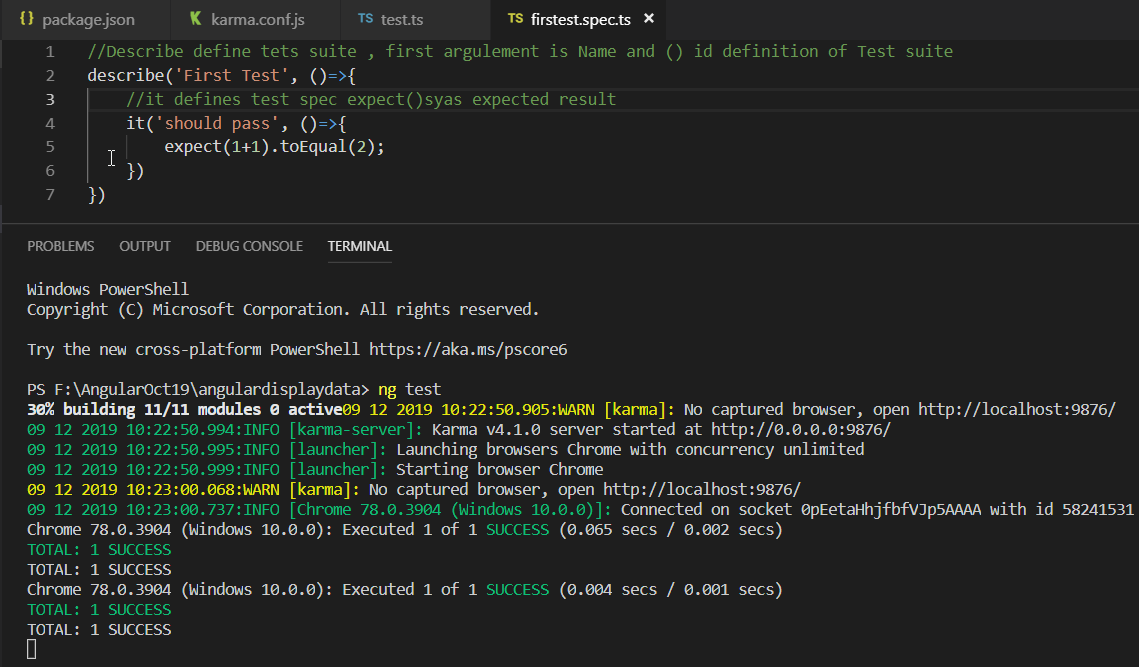


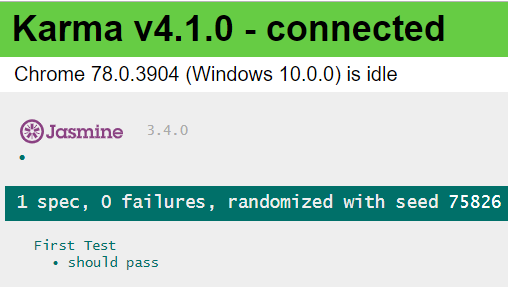
When you run project it reads test.ts and then read line 18 which says go for spec.ts file so ites app.component.spec.ts file.

## Test Case in Angular

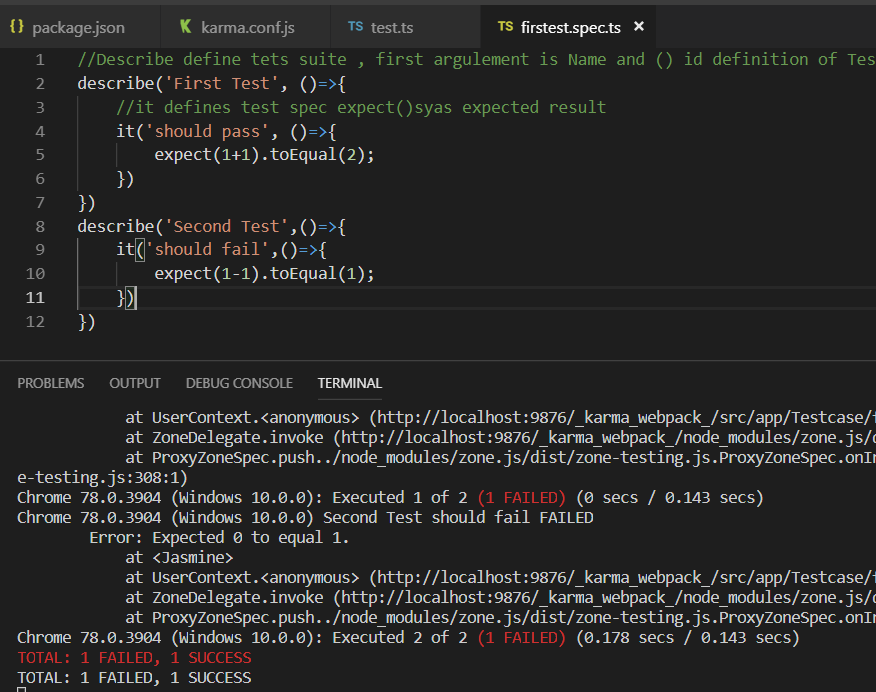
1. Write test spec in angular to get 2 after addeding 1+1.
2. Delete app.componenet.spec.ts file.
3. Create a folder called TestCase/FirstTest.ts

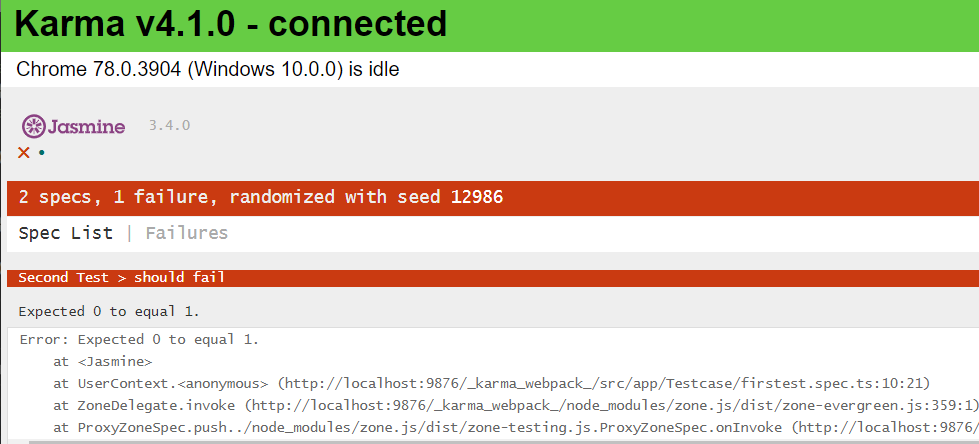


1. 



Lab: Write a Test Case to failed test case 1-1 =1





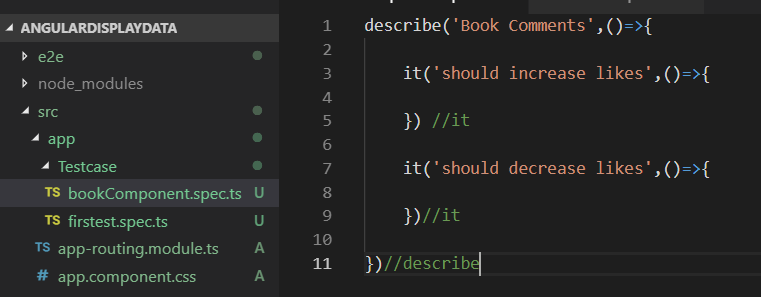
## Arrange Act Assert

The AAA (Arrange, Act, Assert) pattern is a common way of writing unit tests for a method under test.

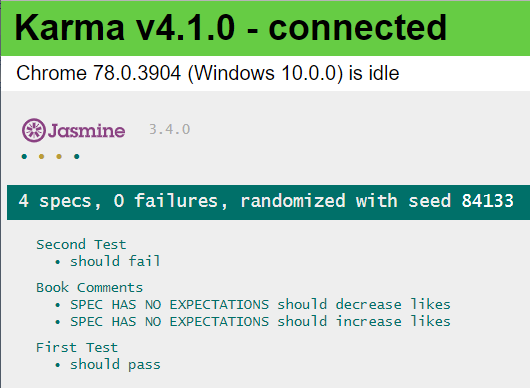
* The **Arrange** section of a unit test method **initializes objects and sets the value of the data that is passed to the** method under test.
* The **Act** section invokes the method under **test with the arranged parameters.**
* The **Assert** section verifies that the action of the **method under test behaves as expected.**

Lab: Create a new test case as BookComponent.spec.ts and define two test cases for increase likes and decrease like with in the test suit as “ Book Comments”.

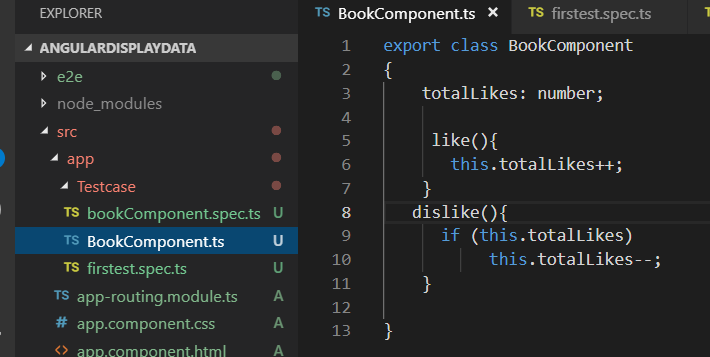
1. Create a bookComponent.ts spec file

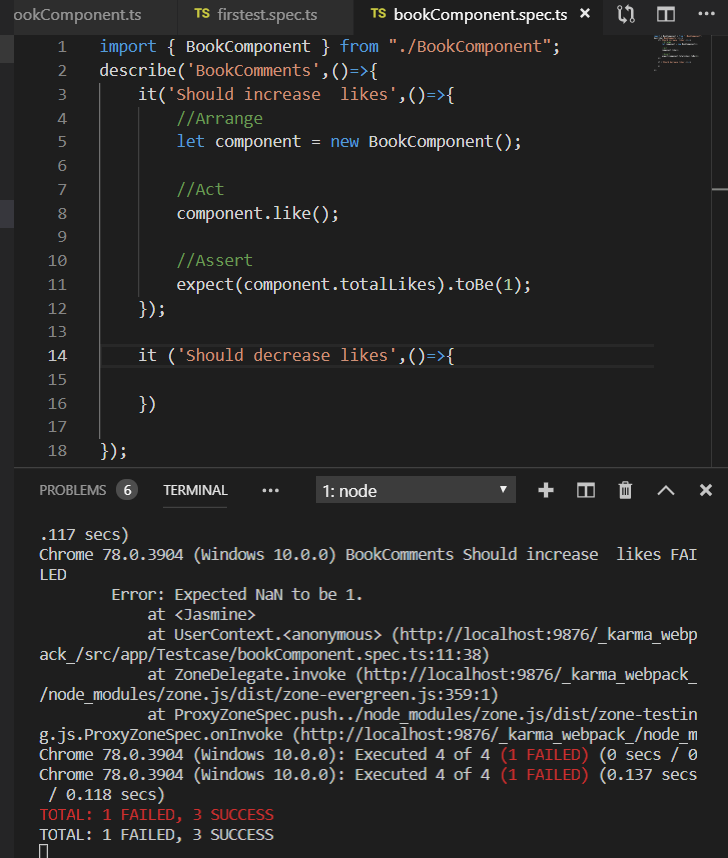
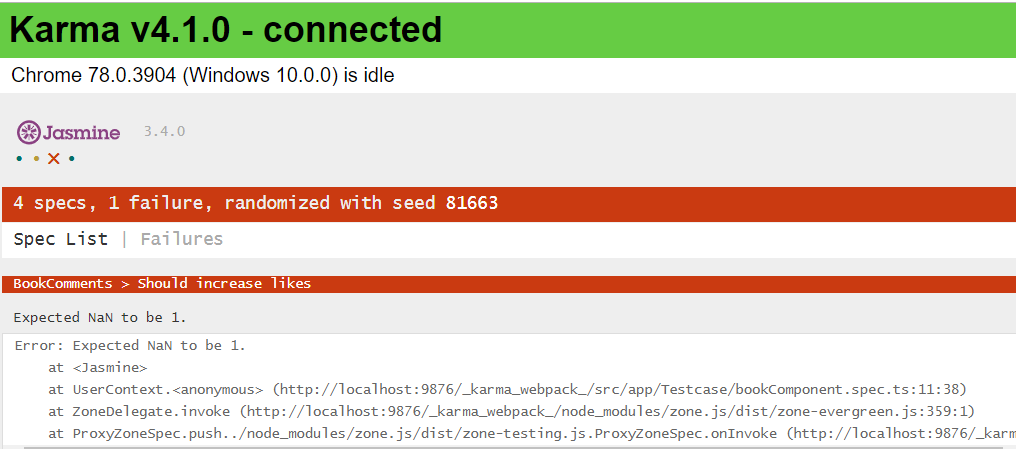


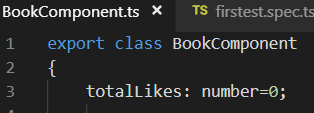
1. Don’t define the body of test case and see effect on browser. The Browser shows “Spec has not expectations”



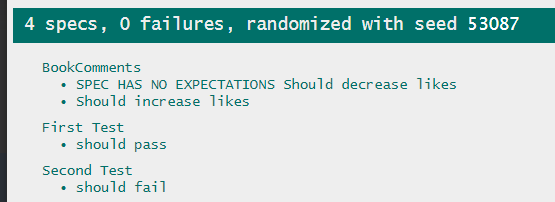
1. Define a BookComponent for a totalLike property as number



1. Define test case to increment the totalLikes property
2. 
3. 
4. It fails because totalikes is number but not initialized so it shows NaN.so lets initialized as 0.

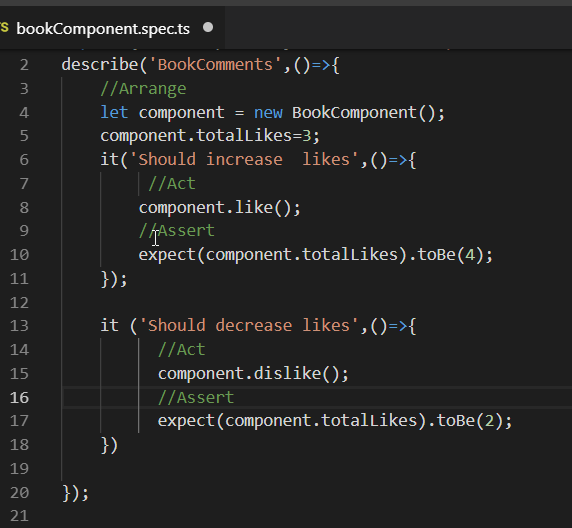




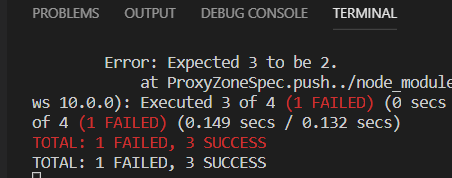
1. 

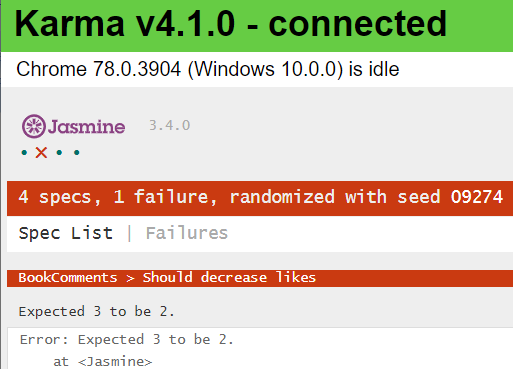
As per the best practice the logic to be in spec file only do the following changes in book.component.spec.ts

Secondly, totallikes would be used by two specs so instead of redefingin it in dislike () we can the initialization part can be taken out.

’’

Review result





**Reason to fail - As it is executing sequentially, It increment to 4 then decrement to 3 so expected result is not 2. So we need to declare component within describe outside the test spec.**

### Introduce Setup and Tear Down

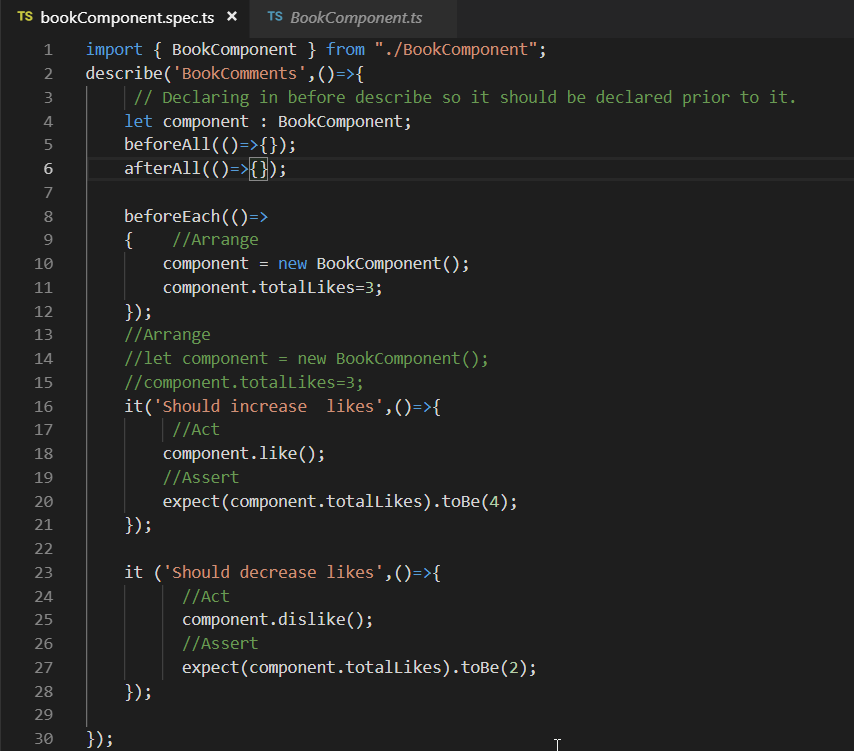
Setup is Before and After is tear down

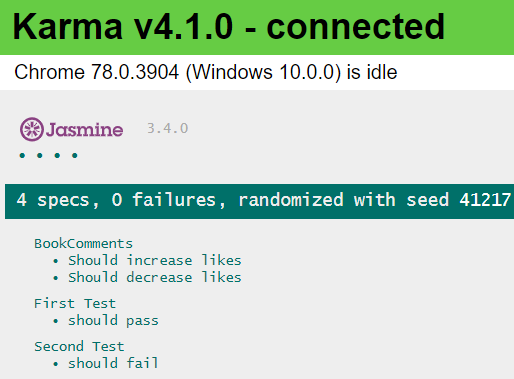
beforeAll() – Execute before describe() - setUp

afterAll()- Execute after describe() – Tear Down

beforeEach()- Execute before each test spec - SetUp

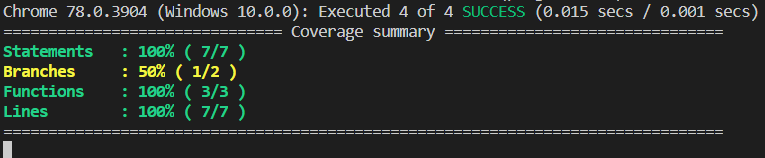
afterEach() – Executes after each test spec – TearDown

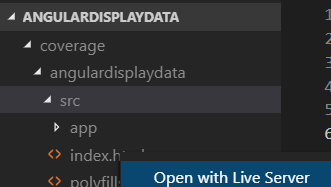
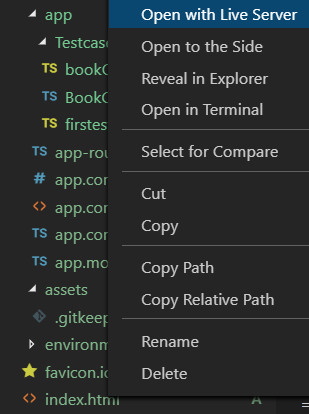
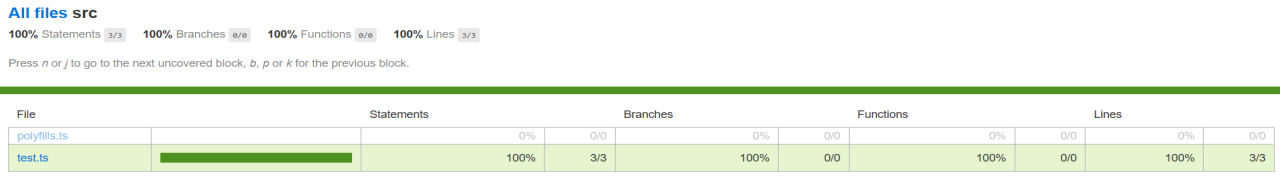
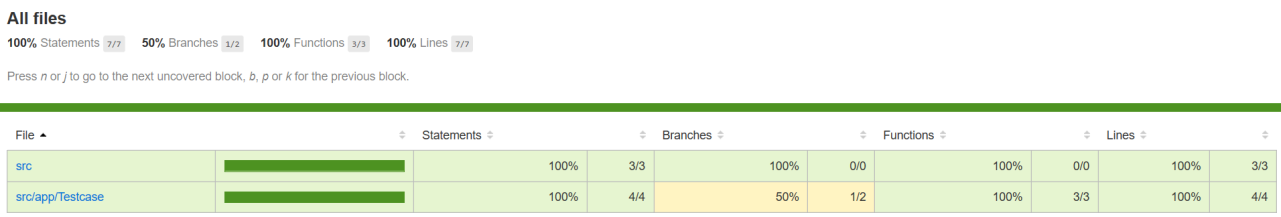




Lab: Evaluate the Code Coverage test for above exercise

* 1. 

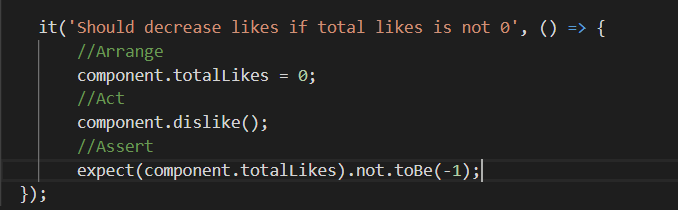


* 1. It will generate the coverage folder -> angulardisplay->src->index.html
  2. Add plug in Live server – to get updates the code coverage report on change in angular.spec.ts
  3. Open index.html in Live server @5500
  4. 
  5. 
  6. Click on All files
  7. 
  8. Click on src/app/testcase

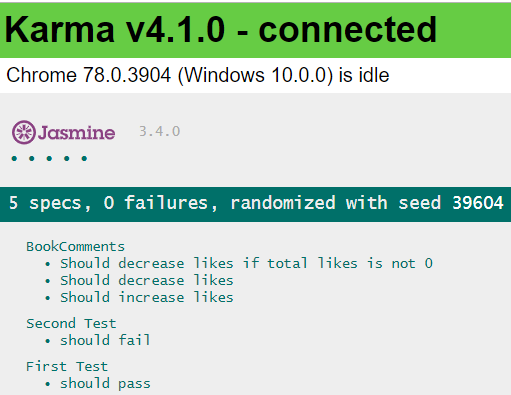


* 1. E stands for Else part not covered so 50%.
  2. 

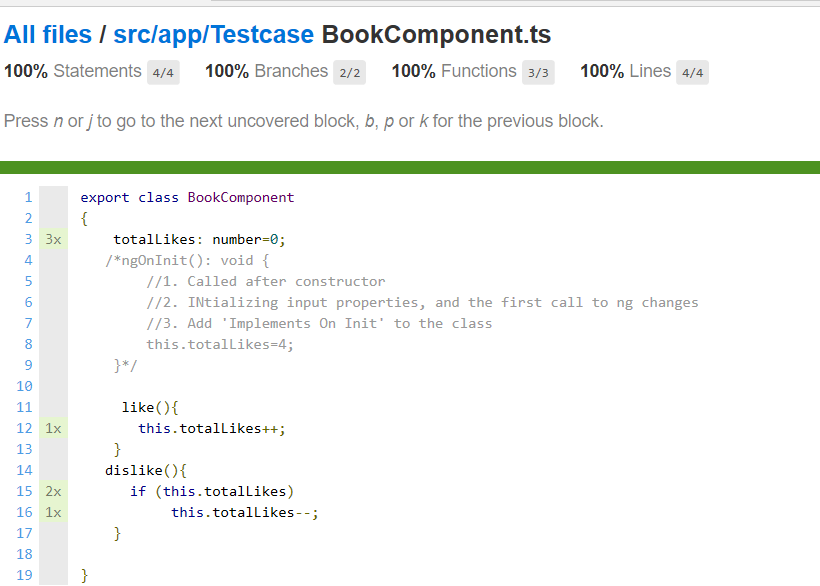
Lab: Add another testcase to dislike only if totallikes are 0.

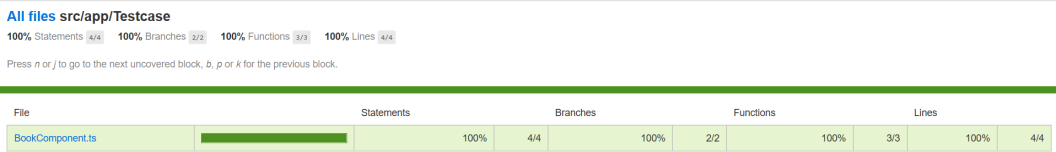


* Ng test—code-coverage



Now the test cases are 100% covered

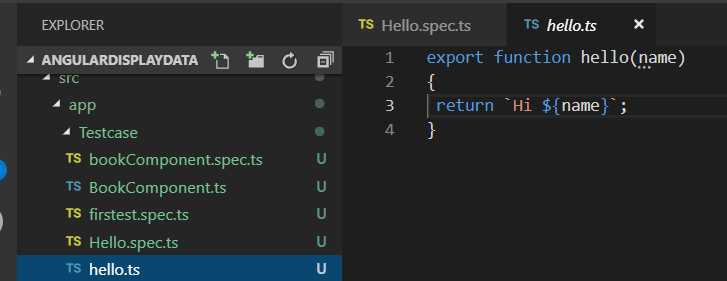


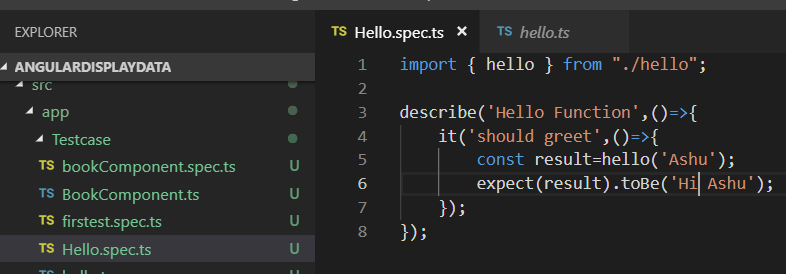


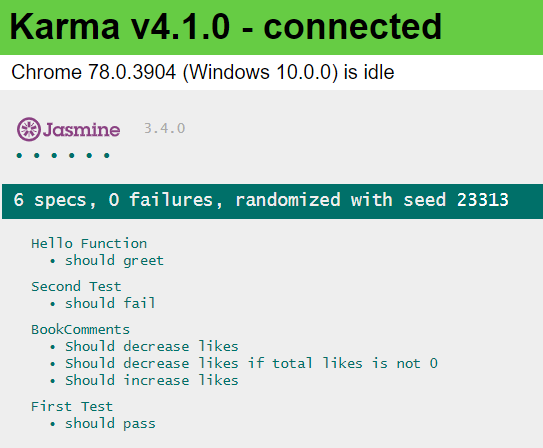
# No Fragile Test

 A **Fragile Test** is a **test** that works - most of the time. A **test** that fails suddenly, even though the tested code is still correct, is called a **Fragile Test**. **Fragile Tests** can undermine everything you worked for so hard when writing your **test** suite.

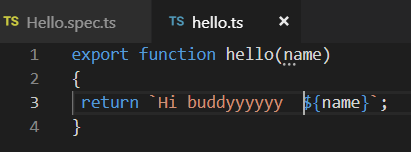
# Lab: Write a test case for hello function which return hello user and test case should verify ‘Hi User’.

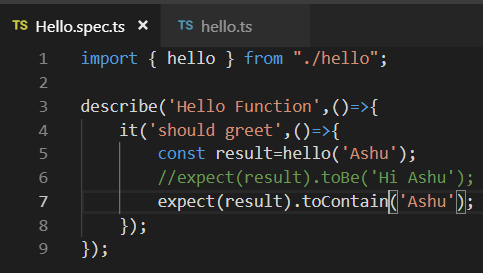




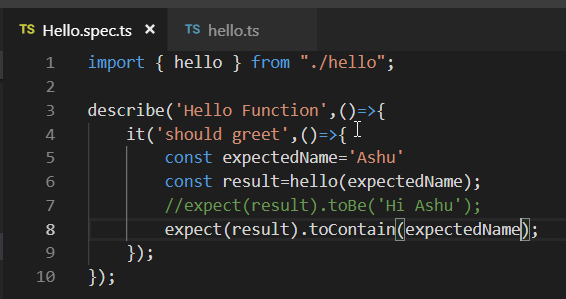


If tomorrow a developer changes Hi to Hi buddy test case may fail. To use toContain() rather thatn toBe().





Note: Test case should not be written to be modified again and again so storing name in a variable is better option.



# Appendix

## Code Coverage

## What is Code Coverage

Code coverage is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determining a quantitative measure of code coverage.

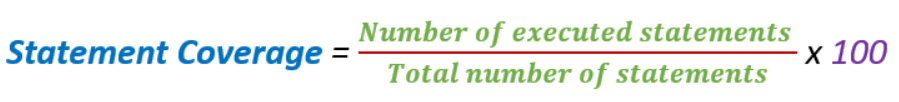
Code Coverage = (Number of lines of code exercised)/(Total Number of lines of code) \* 100%

## Why Code Coverage?

* It helps you to measure the efficiency of test implementation
* It offers a quantitative measurement.
* It defines the degree to which the source code has been tested.

## Types Of Code Coverage

##### Statement Coverage

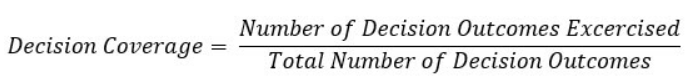
* Involves execution of all the executable statements in the source code at least once. It is used to calculate and measure the number of statements in the source code which can be executed given the requirements.
* 

Example:

|  |  |
| --- | --- |
|  |  |
| Statement Coverage: 5/7 = 71% | Statement Coverage: 6/7 = 85% |

##### Decision Coverage

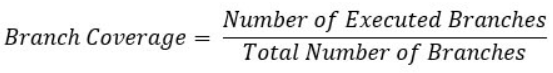
Reports the true or false outcomes of each Boolean expression.

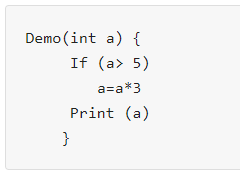


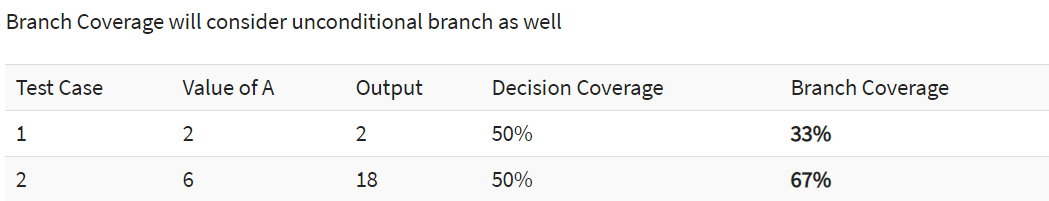
|  |  |
| --- | --- |
| Value =2 | Value =6 |
|  |  |
|  |  |

##### Branch Coverage

Every outcome from a code module is tested. It helps you to ensure that every possible branch from each decision condition is executed at least a single time.





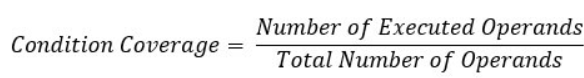


**Advantages:**

* Allows you to validate-all the branches in the code
* Helps you to ensure that no branched lead to any abnormality of the program's operation
* Branch coverage method removes issues which happen because of statement coverage testing
* Allows you to find those areas which are not tested by other testing methods
* It allows you to find a quantitative measure of code coverage
* Branch coverage ignores branches inside the Boolean expressions

##### Toggle Coverage

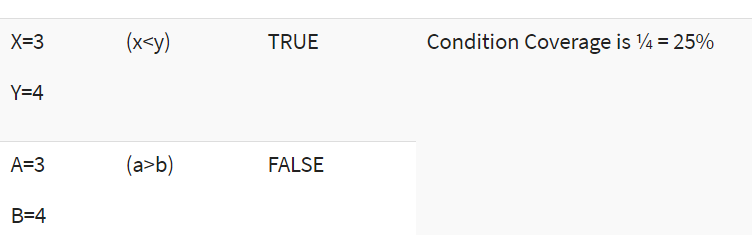
Conditional coverage or expression coverage will reveal how the variables or subexpressions in the conditional statement are evaluated





For the above expression, we have 4 possible combinations

* TT
* FF
* TF
* FT



##### FSM Coverage

## Unit Testing Vs Integration Testing