Karma & Jasmine framework

Karma Framework :

- Karma is a testing automation tool created by the Angular JS team at Google.
- Karma is an open-source tool.
- It provides a **Test runner**.
- Karma is a tool made on top of Node Js to run javascript test cases
- Karma allows us to execute the test on any browser

Installation of Karma :

1. npm install karma karma-chrome-launcher karma-jasmine
2. npm install karma-cli
3. karma –init
4. KARMA start (it will execute the above test file)

Reference :

https://www.guru99.com/angularjs-testing-unit.html#5

Testing in Angular :

→ The Angular CLI downloads and installs everything you need to test an Angular application with Jasmine testing framework

→ **ng test**

The `ng test` command builds the application in *watch mode* and launches the Karma test runner

Jasmine framework :

→ Jasmine is a behavior-driven development framework for testing JavaScript code.
→ It does not require a DOM.

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.

Ex:

```
describe("suite definition ", ()=>{

    it("spec defination", ()=>{})
    it("spec defination", ()=>{})

})
```

How to Exclude Suites :

You can temporarily disable a suite using the `xdescribe()` function. you can quickly disable your existing suites by simply adding an **x** to the function.

Ex:

```
xdescribe()
```

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.

Ex:

```
it(“Spec definition”, ()=>{

        expect(true).toBe(true)
        expect(false).toBe(false)
})
```

How to Exclude Specs:

Just like suites, you can also exclude individual specs using the **`xit()`** function which temporarily disables the `it()` spec and marks the spec as pending.

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 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.

Built-In Matchers :

- **`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.
- **toBeGreaterThan()** for a testing actual value greater than expected value.
- **toBeGreaterThanOrEqual()** for testing the actual value greater than or equal value.
- **`toBeFalsy()/toBeTruthy()`** for testing for falseness/truthfulness etc.

toBe() vs toEqual() :

→ for primitive datatypes toBe() will be useful.
→ for primitive and non primitive datatypes toEqual() will be useful. (It checks deep equality.)

Ex:

```
it( “toBe vs toEqual ”, () => {
        let a = 4;
        let b = [ 1, 2, 3];
        expect(a).toBe(4)              // pass
        expect(a).toEqual(4)           // pass
        expect(b).toBe( [ 1, 2, 3 ] )   // fail
        expect(b).toEqual( [ 1, 2, 3 ] )       // pass
})
```

toBe(true) vs toBeTruthy() vs toBeTrue() :

toBe() :

```
function toBe() {
 return {
 compare: function(actual, expected) {
return { pass: actual === expected };
 } };
}
```

→ It performs its test with **===** which means that when used as expect(foo).toBe(true), it will pass only if foo actually has the value true. Truthy values won't make the test pass.

toBeTruthy() :

```
function toBeTruthy() {
 return {
        compare: function(actual) {
         return {
                pass: !!actual
          }; } }
}
```

toBeTrue() :

```
function toBeTrue(actual) {
        return actual === true || is(actual, 'Boolean') && actual.valueOf();
 }
```

DebugElement :

Is an angular class that contains all kinds of references and methods relevant to investigate an element as well as a component.

fixture.debugElement.query(By.css(“heading”))

→ By.css(“ .class ”) ⇒ to get elements with class

→ By.css(“ #id “”) ⇒ to get elements with id

Native Element :

Returns a reference to the DOM element.

TriggerEventHandler :

Is a function that exists on angular’s debug element.

Code coverage :

Code coverage is a term used in software testing to describe how much program source code is covered by a testing plan. Developers look at the number of program subroutines and lines of code that are covered by a set of testing resources and techniques.

We can approach 2 ways for testing code coverage.

1. ng test –code-coverage (run in command line)
2. Second approach as mentioned below

⇒
```
"test": {
  "options": {
    "codeCoverage": true
  }
}
```

=> ng test