How to Test JavaScript

TDD/BDD with Jasmine and Karma

tl;dr

- Unit testing is the cornerstone of TDD
- TDD = Red, green, refactor
- Jasmine is a unit testing framework for JavaScript.
- It provides a testing framework with a runner
- You can group your tests into modules with Jasmine
- Jasmine has assertions (toBeEqual, toBe, etc.)

A manual test script

Test if new passwords don't match, error message shows.

1. Go to /changePassword
2. Put old password in box 1
3. Put new password in box 2
4. Put different new password in box 3
5. Hit button
6. Did error message show and say something about passwords did not match?
7. If yes, test passes. If no, test fails.

Testing by hand stinks!

- Team stops coding to write test scripts and then run those scripts
- Super-repetitive!
- Boring!
- Waste of time and talent!
- Automated testing is the way to go!

Thus, many tools have been developed

- Test runners
  - The UI for running a test and seeing the results
  - eg. qUnit, Jasmine, Mocha, Karma
- Testing frameworks
  - Organize/group the tests and hold the tests themselves
  - eg. qUnit, Jasmine, Mocha, Cucumber
- Assertion libraries
  - Hold the different possibilities of conditions that can be looked for
  - eg. qUnit, Jasmine, Chai, Should
- Other tools to help the above
  - eg. TestDouble, Sinon, Blanket, Istanbul, PhantomJS, Casper

These all use assertions

```
expect(getName()).toEqual("Jo")
```

Quicker and simpler than writing a bunch of if-else statements

```
let actual = getName();
if (actual !== "Jo") {
  throw new Error(
    `Expected name to be equal to Jo but got ${actual}`);
}
```

What is unit testing?

A unit test tests one unit of work

- One requirement for one method
- They are:
  - Isolated from other code
  - Isolated from other developers
  - Targeted
  - Repeatable
  - Predictable

What is NOT unit testing? ... Anything else!

- > 1 requirement
- > 1 method
- > 1 project
- > 1 system
- Affects > 1 developer

Next, what is TDD?

Red, green, refactor

The red phase

You want to write positive tests and negative tests

- Negative tests involve values that are outside acceptable ranges.
  - They should fail
  - You're testing to make sure that they do
- Positive tests are ones that should pass

The green phase

The refactoring phase

Once finished you pick up a user story and start again writing a new failing test

One of the biggest motivations for testing is **engineering velocity**.

It may seem that writing tests slows down the development process, but this only holds in the short term. Without automated tests, projects can only grow so much before our ability to deliver grinds to a halt.

It enables new contributors to push code without worry of breaking anything.

It battles code rot by alleviating fear of drastic refactoring.

It enables faster releases with more confidence.

It allows us to fix issues in production continuously without waiting for manual regression.

Let's see how to do this with Jasmine

To get it, install it with npm and create a config file called jasmine.json

Jasmine will interview you to create that jasmine.json config file

Jasmine locates your tests by itself and run them

describe() creates groupings of tests called "modules"

```
describe("Car", () => {
// Tests of car things go here
});

describe("Person", () => {
// Tests of people things go here
});
```

it() defines the tests themselves

Syntax

```
it(title, testFunction);
```

Example

```
it("can accelerate", function () {
    let s1 = car.speed = 10;
    car.accelerate(35, 5);
    expect(car.speed).toEqual(45);
});
```

Jasmine's built-in assertions have expectations and matchers

- It'll always be a form of:

```
expect(value).matcher(mayOrMayNotHaveValue)
```

- The matcher throws if its requirement isn't met.

The matchers are pretty simple

- toBe(actual)
- toBeClose(actual, precision)
- toBeDefined() | toBeUndefined()
- toBeFalsy() | toBeTruthy()
- toBeGreaterThan(actual) | toBeLessThan(actual)
- toBeGreaterThanOrEqual(actual) | (and less)
- toContain(actual)
- toEqual(actual)
- toMatch(regex)
- toThrow()
- ... and a few more

The most basic check is toBeTruthy()

- If the parameter passed is truthy, it passes
- If not, it fails

Syntax

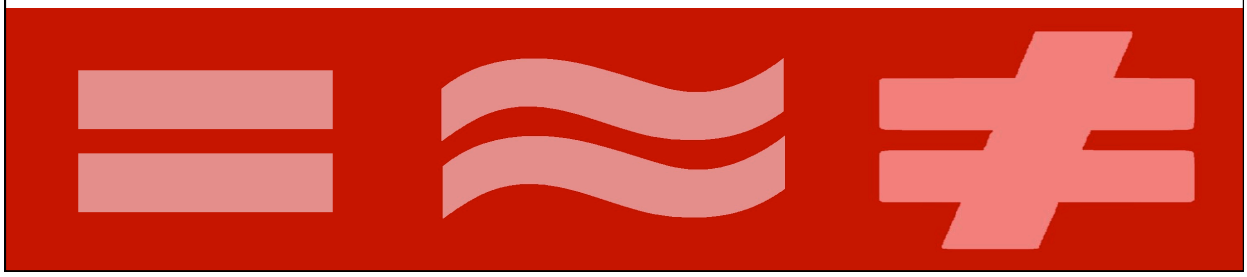
```
expect().toBeTruthy()
```

Example

```
QUnit.test("Death Star is really big", function () {
  expect(this.deathStar.getSize() > 1e25).toBeTruthy();
}
```

How to test for equality

toBe(a, e)	toEqual(a, e)
Makes sure the actual value and the expected value are the same. Like "===".	Like toBe() but for objects. Checks all values at all depths.



toThrow() makes sure an Error was thrown

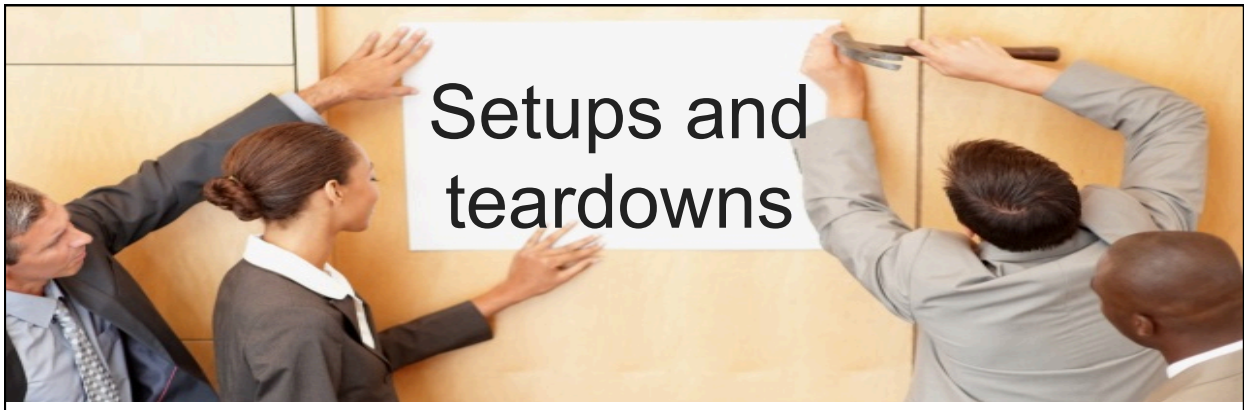
Syntax

- expectation.toThrow(function[, expectedError][, expectedMesg]);

Example

```
it("should throw when a jedi attacks a Jedi", function () {
    expect(function () {
      var luke=new Jedi('Luke Skywalker');
      var mace = new Jedi('Mace Windu');
      luke.attack(mace);
    }).toThrow(/traitor/);
});
```

- The highlighted function above is expected to throw an error with a message that contains the regular expression "traitor".
- If not, the unit test fails.




```
describe(name, function () {
 beforeEach(function () { /* Do setup things here. */})
 afterEach(function () { /* Do teardown things. */} )
 beforeAll(function () { /* Do setup things here. */} )
 afterAll(function () { /* Do teardown things. */} )
}
```

Shared variables

- You can also have shared variables if you define them in the lifecycleObject:

```
define("Sith Object Tests", function () {
 let master; let vader;
 beforeEach: function () {
    master= new Sith('Darth Sidius');
    vader = new Sith('Darth Vader');
    master.setApprentice(vader);
  }
});
```

Karma

Automated browser testing

Karma is a browser test runner

- It creates a Node web server to serve ...
  - the files under test
  - the spec files
  - Karma's results panel html
- It fires up the browser to run the tests in
  - can do it in >1 browser at a time (Chrome, FF, Edge, PhantomJS)
- Bottom line: If you want to run your tests in a browser, you need Karma

To run tests, you will tell Karma ...

1. What framework you're using (eg. Jasmine)
2. Where the Jasmine tests are located
3. What browsers you want to run the tests in

- And, like Jasmine, Karma will ask you these things in an interview and save it in a config file
- Then, just run it.

tl;dr

- Unit testing is the cornerstone of TDD
- TDD = Red, green, refactor
- Jasmine is a unit testing framework for JavaScript.
- It provides a testing framework with a runner
- You can group your tests into modules with Jasmine
- Jasmine has assertions (toBeEqual, toBe, etc.)