Back to Week 2

JS | Testing & Jasmine

LESSON

Learning goals

After this unit, you will be able to:

- understand what testing is and why it is important for your apps,
- understand the structure of a test,
- understand what Unit Testing and TDD are,
- create your first tests to *protect* your code.

Introduction

Until now, we have been using pre-written tests. That is not really the TDD approach. Writing tests has a lot of extra benefits, especially when you write them yourself. One argument that is often used against writing tests is that it supposedly costs too much time. A very common counter-argument is, that it maybe costs some additional time now, but that in the long run, it pays off for the project. Although that is proven to be true, it doesn't stop there. Writing tests save you time right away because you automate a part of your own development process as well! In addition to that, it forces you to divide the program as you develop it into nice bite-size chunks (lots of very simple functions that will when implemented together make your whole program). So on top of a **stable product in the long run**, you also write ** better-structured code**.

Let's learn more about testing.

What is testing?

> Software testing is a process of executing an application to validate and verify that it meets the business and technical requirements and works as expected.

Testing is a process, not a single activity. So the process of designing tests early at the beginning of the development and the product's life cycle can help to prevent deficiencies in the code or product design.

Our applications are, in its core, just a bunch of functions. What we want to achieve with tests is to make sure these functions will perform how we want them to perform and return expected outputs, no matter what kind of input they receive.

Why should I test?

Let's start with our usual workflow, without tests. Yoult goes - code a little, execute the code, check if it works in the console, or by using the debugger. If it doesn't work, you go back and you try to fix it. Then you go back to the console or debugger. In case it works, you move on to the next bit of code. Gradually you build up your program like this. There is this other group of people that writes the whole program at once and then execute it at the end and hopes that it works. Most likely, it will not work and they will have to look for a needle in a haystack to debug their own code. Don't ever be this other group of people!

Let's be realistic. When we code, even if we are code Jedis, there is always a chance that we will introduce errors. Assuming that you can code the whole app without a single mistake, or that you can predict all the consequences of

an error, is just simply the wrong attitude. A good set of tests is the best safety net we can provide for our apps.

Tests prove that your code actually works in every situation in which it should work. Even when you are improving the design or creating new features, you can change your current code without breaking what already works.

Unit testing

Unit testing allows you to build up a battery of small tests that verify fine-grained bits of your code, especially edge cases. This is especially useful in JavaScript, where your application needs to run the same way on different browser platforms.

The core idea of unit testing is to test a function's behavior when giving it a certain set of inputs. You call a function with certain parameters and check you got the correct result.

This has a couple of huge advantages:

- **You always test your code in the same way**. Goodbye self-doubt like: "Did I call it the same way? Is it different because the situation is different or did the behavior of my code change unexpectedly?"
- **With every new bit of code you are testing all of your previous code again. Automatically!** What does this mean? The situations like "Hmm that used to work before, where did it break?" will be passed since with TDD you KNOW where and when it broke. You don't have to go through the whole codebase again to find the needle.
- **Usually manual testing takes a lot of steps and therefore costs you the time**. Let's say you are making the Viking assignment without tests. If you want to test the War class, you first have to create Vikings, Saxons, then add them to the War instance and make them fight. Just automate all of this with tests! You have to do the manual testing anyways.

Jasmine

Jasmine is an automated testing framework for JavaScript. It is designed to be used in **BDD** (**B**ehavior-**D**riven **D**evelopment) programming, which focuses more on the business value than on the technical details.

Getting Jasmine

You can run Jasmine from the browser using the standalone distribution. It contains everything you need. To get it, download the latest version in a zip file from the official **release page**.

Unzip it and empty the `/spec` and the `/src` directories (those files are examples and we will replace them with our code).

Write a function

We will create a `greeting.js` file under our `/src` folder and write our first JavaScript function to say hello to all Ironhackers.

```
// src/greeting.js
function greeting() {
  return 'Hello Ironhackers!';
}
```

Explain this code

Easy, right? Now comes the test.

Write a test

The function above is quite simple, but we can begin to think a bit as a *tester* and notice that every time we call this function, no matter in which context, it will always return a string with the same value.

Let's create a `greeting.spec.js` file in the `/spec` folder and its test to verify this happens:

```
// spec/greeting.spec.js

describe('The function greeting', () => {
  it('should greet all Ironhackers', () => {
    expect(greeting()).toEqual('Hello Ironhackers!');
  });
});
```

Explain this code

Let's explain the code above a little bit before witnessing the magic in it.

- `describe (string, () => {})`: This is called a **suite**, a group of tests for a component of our application (could be a class or a bunch of functions). It looks like an anonymous function because it is actually an anonymous function. However, the string we pass as the first argument is a name or title for a spec suite - usually what is being tested.

- `it(string, () => {})`: This is called a **spec** (from specifications) and it contains one or more expectations that test the state of the code. The first string is the title and describes in English the expected behavior. Each suite can hold several specs according to the outputs we are expecting.

- `expect(function.matcher(output)`: This is the actual **expectation** that will test the code. This is an assertion that could be `true` or `false`. These are built with the function `expect,` and it is chained with a **Matcher** function. This matcher function should receive the expected output according to its comparison. We will learn more about Matchers in a bit.

👓 A spec with one or more **false** expectations is a **failing spec.**

The SpecRunner HTML

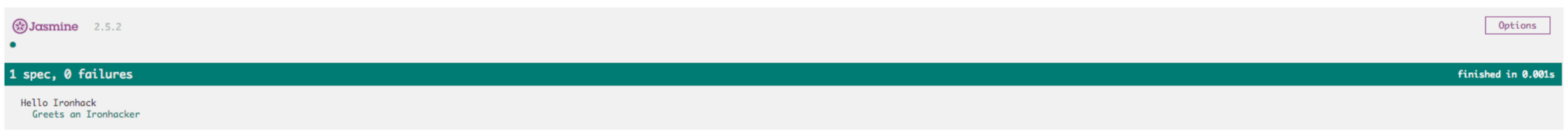
In the root of our Jasmine folder is a file called `SpecRunner.html`. Let's take a look at this file in the text editor.

In the `head` section of the `HTML` file, we will include both our `js` files and our `spec` files.

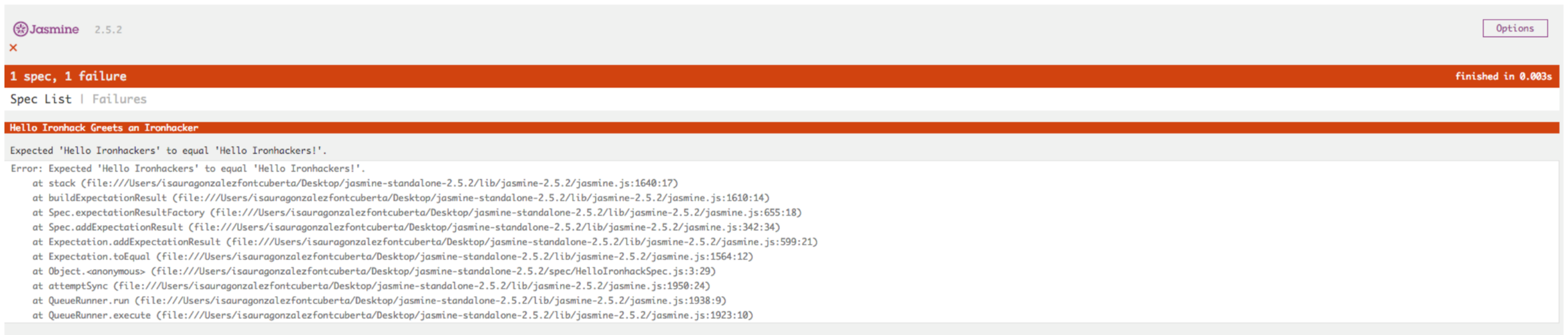
```
<!-- include source files here... -->
<script src="src/greeting.js"></script>

<!-- include spec files here... -->
<script src="spec/greeting.spec.js"></script>
```

Explain this code

Now, open the `SpecRunner.html` file in your browser.



If the output of the matcher is `true` for every test, your HTML should look like the picture above. Change your function to return a different string and reload the page. Jasmine complains!!! Read the result of the execution of your test to gather information about how to fix it.



But that function is lame. What happens if we want to develop a method that receives parameters? Let's use TDD to create a more complex function.

TDD

TDD (stands for **T**est **D**riven **D**evelopment) is an approach to writing software where you write tests **before** you write application code. It is composed of three steps:

1. Write a test and **watch it failing**. This methodology is applied correctly when we make sure the test fails, and we have to be forced to fix it.
2. Write the simplest, easiest possible code to make the test pass.
3. Refactor and simplify the application code without breaking the test.

These steps are followed once and again in a loop until we finish the method. To remember them, you can use the TDD mantra: *red, green, refactor*.

A guided exercise

Using TDD, let's create a calculator with a function that receives an array of numbers and returns the addition of every element in it. Let's create files we will be working in:

- `src/calc.js` and
- `spec/calc.spec.js`.

In the `SpecRunner.html`, add these two files to be able to see the tests:

```
<!-- include source files here... -->
<script src="src/calc.js"></script>

<!-- include spec files here... -->
<script src="spec/calc.spec.js"></script>
```

Explain this code

Using the methodology, first, we should create the test:

```
// spec/calc.spec.js

describe('The function sum() used in the Calculator', () => {
  it('should be a function', () => {
    expect(typeof sum).toBe('function');
  });
});
```

Explain this code

Now, let's write the simplest possible implementation that makes the test pass.

```
// src/calc.js

function sum(array) {}
```

Explain this code

Even if it looks a little bit awkward at first and the implementation looks trivial, it is pretty useful. Let's write a new test inside the *describe* above. The first thing we want to do is to take care of the edge cases. One thing we can start with is to make sure that the function gives a correct result if there are no numbers passed to function (by default, the calculator should return zero before user inputs any numbers to do calculation on them):

```
// spec/calc.spec.js

describe('The function sum() used in the Calculator', () => {
  it('should be a function', () => {
    expect(typeof sum).toBe('function');
  });

  it('should return 0 for an empty array', () => {
    expect(sum([])).toBe(0);
  });
});
```

Explain this code

And now the implementation:

```
// src/calc.js

function sum(array) {
  // if (!Array.isArray(arr)) return 0;
  if (array.length === 0) return 0;
}
```

Explain this code

The next thing we could take care of is making sure the user provides some data (numbers) before requesting some mathematical operation to be done on them:

```
// spec/calc.spec.js

describe('The function sum() used in the Calculator', () => {
  it('should be a function', () => {
    expect(typeof sum).toBe('function');
  });

  it('should return 0 for an empty array', () => {
    expect(sum([])).toBe(0);
  });

  it('should throw an error if no parameter is provided', () => {
    expect(() => {
      sum();
    }).toThrow(new Error('No parameter provided'));
  });
});
```

Explain this code

Now, we should add some code to our `sum()` function, making sure it passes this test:

```
// src/calc.js

function sum(array) {
  if (arr === undefined) {
    throw new Error('No parameter provided');
  }

  // if (!Array.isArray(arr)) return 0;
  if (array.length === 0) return 0;
}
```

Explain this code

The next thing we should take care of is to make sure that our `sum()` function always returns a number. That is pretty logical, but, we have to make sure to take care of all the possible cases.

```
// spec/calc.spec.js

describe('The function sum() used in the Calculator', () => {
  it('should be a function', () => {
    expect(typeof sum).toBe('function');
  });

  it('should return 0 for an empty array', () => {
    expect(sum([])).toBe(0);
  });

  it('should throw an error if no parameter is provided', () => {
    expect(() => {
      sum();
    }).toThrow(new Error('No parameter provided'));
  });

  it('should return a number', () => {
    expect(typeof sum([1, 1])).toBe('number');
  });
});
```

Explain this code

After this point, we should proceed to write the algorithm to calculate the sum. The sum should always reflect the correct addition of all the elements passed to the `sum()` function. Let's wrap this up into a test and then let's proceed to write the algorithm:

```
// spec/calc.spec.js

describe('The function sum() used in the Calculator', () => {
  it('should be a function', () => {
    expect(typeof sum).toBe('function');
  });

  it('should return 0 for an empty array', () => {
    expect(sum([])).toBe(0);
  });

  it('should throw an error if no parameter is provided', () => {
    expect(() => {
      sum();
    }).toThrow(new Error('No parameter provided'));
  });

  it('should return a number', () => {
    expect(typeof sum([1, 1])).toBe('number');
  });

  it('should return the correct value', () => {
    // if array with one element is passed, sum should be equal to  that element
    expect(sum([1])).toBe(1);
    expect(sum([1, 2])).toEqual(3);
    expect(sum([4, 5, 6])).toBe(15);
    expect(sum([0, -2])).toBe(-2);
    expect(sum([1, 2, 3, 4, 5])).toBe(15);
  });
});
```

Explain this code

And `sum()` gets updated with the following:

```
// src/calc.js

function sum(array) {
  if (array === undefined) {
    throw new Error('No parameter provided');
  }

  // if (!Array.isArray(arr)) return 0;
  if (array.length === 0) return 0;

  return array.reduce((acc, value) => acc + value, 0);
}
```

Explain this code

> 💣 **Remember to ALWAYS watch the test fail before fixing it**.

Check for understanding

Using TDD, create tests, and then function `avg()` to calculate the average of the inputted array.

More Matchers

Jasmine provides tons of matchers to create your expectations. We have collected the most common ones here:

Matcher	Description
`expect(x).toEqual(y);`	Compares objects or primitives x and y and passes if they are equivalent
`expect(x).toBe(y);`	Compares objects or primitives x and y and passes if they are the same object
`expect(x).toMatch(pattern);`	Compares x to some string or regular expression pattern and passes if they match
`expect(x).toBeDefined();`	Passes if x is not undefined (expects actual value to be defined).
`expect(x).toBeUndefined();`	Passes if x is undefined
`expect(x).toBeNull();`	Passes if x is null
`expect(x).toBeTruthy();`	Passes if x evaluates to true
`expect(x).toBeFalsy();`	Passes if x evaluates to false
`expect(x).toContain(y);`	Passes if array or string x contains y
`expect(x).toBeLessThan(y);`	Passes if x is less than y
`expect(x).toBeGreaterThan(y);`	Passes if x is greater than y
`expect(() => {fn(); }).toThrow(e);`	Passes if function fn throws exception e when executed

Creating custom matchers

Custom matchers help to document the intent of your specs and can help to remove code duplication in your specs. See the Extra Resources section for a guide on how to create them.

Summary

Now you know how important it is to test your code. Also, you know you can apply two different methodologies to cover your code with testing: Unit Testing or TDD. You also know the structure of a test file in JavaScript and created your first tests. Are you ready to apply this to the rest of the bootcamp and your career as a developer?

The complete code example can be found **here**.

Extra Resources

- **JavaScript Testing Framework Comparison: Jasmine vs Mocha**
- **Jasmine Custom Matchers**
- **Jasmine Github**

Mark as completed

PREVIOUS
← LAB | DOM Ironhack Cart

NEXT
JS | Intro to testing with Jasmine →