Learning Objectives

Students will be able to…

- **Identify when unit testing should be used**
- **Define Jest and describe its usage**
- **Install Jest**
- **Create unit tests using Jest**

info

Make Sure You Know

Basics programming language JavaScript, including writing functions with the arrow syntax.

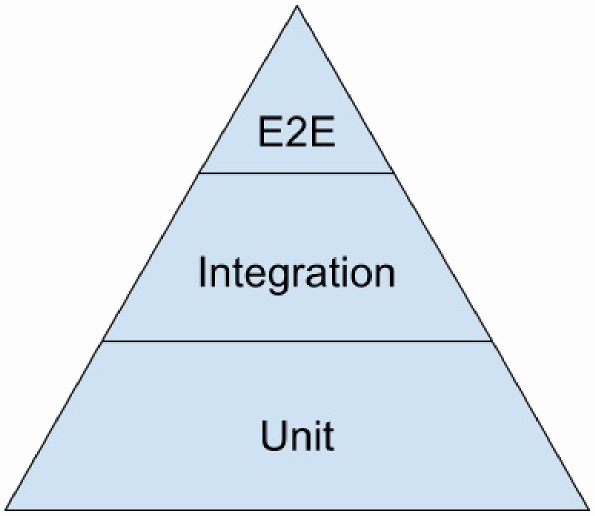
Limitations

While testing can be performed in a variety of ways, this content makes use of the following software:
- Jest framework
- npm package manager
- Node.js

About Unit Testing

Unit Testing is part of the testing pyramid. There are many options of the testing pyramid, but most are: Unit, Integration, and End2End. Other names for these tests include Unit - module testing, Integration - API testing, and End2End - UI testing.

In this assignment we will cover Unit testing, work with the Jest framework in more detail, and write our first tests. Integration and End2End testing will be discussed in more detail in the following modules.



This depicts a pyramid with three levels. The top level is E2E testing, the middle layer is integration testing, and the bottom layer is component testing.

Unit testing is a software testing method in which modules are created, that is, small parts of an application, the behavior of each of which is tested separately. Unit testing is performed during the development phase of an application. A module can be anything, such as a procedure or a function.

Unit testing contains three stages:
- Initialization of a small fragment of the application that you want to test.
- To do this, a method is called that is applied as a stimulus to the system under test.
- Monitoring the behavior of the module under test.

This step-by-step process is also called **AAA** (Arrange, Act, Assert).

Why Unit Testing?

Unit testing is popular due to its many **advantages**. Here are a few of the important ones:
- Unit testing makes development easier to change and maintain code.
- Unit tests can be reused.
- Unit testing speeds up development.
- Unit tests are more robust and run faster in the long run.
- Less costly in terms of time and other resources.
- Unit testing makes debugging easier.
- This approach improves the design of the code and allows it to be refactored.
- Developers can understand what each module makes and look at unit tests to get a basic understanding of the API.
- You can test a selected part of code without waiting for another to complete.

Thus, Unit testing has a lot of advantages and it is about 60% of all testing. But there are also some **disadvantages**:
- Unit testing does not reveal all the bugs in the program.
- Cannot evaluate all execution paths, even in simple programs.
- Mainly focuses on units of measurement and cannot detect integration errors at a broader level.

The general rule is that Unit testing should be done in conjunction with other tests to get more accurate results.

Unit tests can be simple or complex depending on the nature of the object being tested and the tools and strategies used for testing. The only thing you can be sure of here is that Unit testing will make your program more robust and functional than untested applications.

Installing and Using Jest

About Jest

Jest - JavaScript testing framework. It allows to write tests with an acceptable, familiar and functional API, and quickly achieve the results.

Jest is well-documented, requires minimal customization, and can be extended to meet requirements and makes testing easy. It also works with projects using Babel, TypeScript, Node, React, Angular, Vue, and others.

Benefits of Using Jest

There are many advantages of the Jest framework, some of which include:

- Easy to start - Jest allows you to work from the box with no additional configuration for most JavaScript projects.
- Fast and productive - Jest runs previously failed tests first and changes the order in which tests are run based on how long each test runs.
- Exceptions - When tests crash, Jest provides a detailed description of the reason for the crash.
- Easy mocks - Jest uses its own mechanisms to resolve import conflicts in tests, allowing to easily mock any imported object that's outside the test environment.
- Availability of snapshots - Write tests that easily track large objects. Snapshots live either next to your tests or spelled out right in your code.
- API - From `it` to `expect` Jest has a full suite of tools from the box.

For more information you can view the Jest documentation. On the following pages we will try to install the Jest framework, learn the basics, and write some Unit tests.

Installing and Configuring Jest

The first thing we need to do is set up an NPM package. Enter the following command into the terminal:

```
npm init
```

NPM will ask you several questions. We are going to enter a few pieces of information and use the defaults for the rest:
* package name - type `jest_test`
* version - press `ENTER` to use the default
* description - press `ENTER` to use the default
* keywords - press `ENTER` to use the default
* author - press `ENTER` to use the default
* license - press `ENTER` to use the default

NPM will then print the contents of the `package.json` file, which contains all of the information from the above questions. Press `ENTER` one more time to create the `package.json` file

When creating the NPM package, we stated that Jest will be used for testing. We need to now install this package. At the time of writing, version 29.3.1 is the latest version of Jest. Install this version by running the command below in the terminal:

```
npm install --save-dev jest@29.3.1
```

▼ **Node and NPM**

You can check NPM and Node versions in terminal with the commands `npm -version` and `node -version`. It’s good to use one of the latest versions otherwise we will have a problem installing Jest.

Once the installation process is done, enter the following command in the terminal:

```
cat package.json
```

This prints the contents of the JSON file. At the bottom, you should see Jest listed as a development dependency.

```
  "devDependencies": {
    "jest": "^29.3.1"
  }
```

We will write Unit tests in the next pages and we will learn this in more detail. Take a look at the documentation to learn more about getting started with the framework.

Unit Testing Intro

Creating a Unit Test

Now let’s try to create our first unit test. This test is going to check a function that returns the square of a number. Start by importint the `square.js` file and store it in the constant `square`:

```
// Importing a function
const square = require('./square');
```

Use the keyword `test` to create the test. Be sure to use a descriptive string to identify the purpose of the test. The `expect` keyword is followed by a call to the `square` function with `2` as the parameter. Use dot notation to chain the `toBe` method to `expect`. Put the expected value between parentheses. Jest will compare the result of `square(2)` with `4`.

```
// Test our function
test('2 * 2 to equal 4', () => {
    expect(square(2)).toBe(4);
});
```

Copy the command below and paste it into the terminal. Press `ENTER` to run the test.

```
npm run test square.test.js
```

The test will not pass as the function `square` is not yet implemented. We are only showing the example of a failing test. It may seem odd to write the test before the function, but this approach is used in Test-Driven Development (TDD) which we will discuss later.

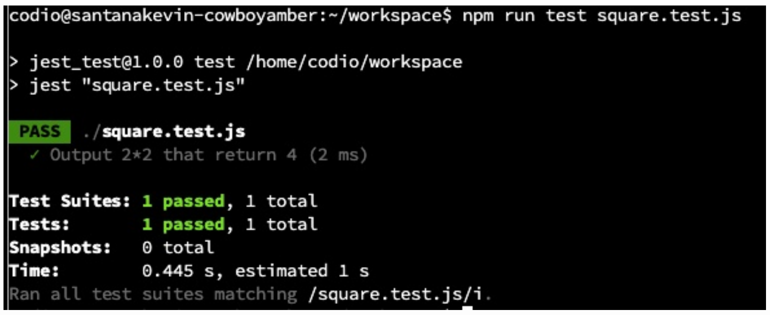
Next, we are going to declare the function `square` in the bottom-left file. Create the function `square` that takes a parameter. Return the number multiplied by itself. Remember, you need to export the function so it can be called by the unit test.

```
function square(x){
    return x * x;
}
module.exports = square;
```

Run the unit test one more time by entering the following command into the terminal:

```
npm run test square.test.js
```

The test should now pass, and the results will look something like this:



```
codio@santanakevin-cowboyamber:~/workspace$ npm run test square.test.js

> jest_test@1.0.0 test /home/codio/workspace
> jest "square.test.js"

 PASS  ./square.test.js
  ✓ Output 2*2 that return 4 (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.445 s, estimated 1 s
Ran all test suites matching /square.test.js/i.
```

The image depicts a passing Jest test.

▼ **Running all of the tests at once**

As you develop robust tests for all of the components of your system, you find running each test individually to be tiresome. The following command will run all of the tests at once.

```
npm run test
```

If you were to enter this command now, Jest will run test files that have yet to be completed. Some tests will fail.

Now that the function is defined, and we check how it works by means of a matching. The matching is done with the `expect()` function and the evaluator. In this case, it is a check for exact equality `toBe()`. There are others like `toEqual` , `toStrictEqual` , `toBeNull` , `toBeUndefined`, `toBeTruthy` and many more. We will look at some of them in the following pages. You can find more information at the Jest website about using-matchers.

Methods toBe, toEqual and others

Basic Methods

On the previous page, we saw how we can chain a using-matcher to `expect` to specify the type of test we want to perform. Here is a closer look at some of the common using-matchers. **Note**, not all of the examples have code snippets.

info

Understanding Jest Output

Lines of code that start with `expect` are test cases. Look at the comments at the end of each test case when you copy and paste the example into the IDE. You will see if it passes or fails. All test cases must pass in order for a test to pass.

You may see error messages in the Jest output. That does not mean something is wrong. Some of these tests were designed to fail.

- **toBe()** - is suitable if we need to compare primitive values or if the passed value is a reference to the same object that is specified as the expected value. Values are compared using `Object.is()`. As opposed to === it gives us a way to distinguish `0` from `-0`, to check if `NaN` is equal to `NaN`.
- **toEqual()** - is suitable if we need to compare the structure of more complex types. It will check every element of the array. And it will do it recursively through all the nesting.

```
test('toEqual with objects - pass', () => {
    expect({ zoo: 'zoo', subObject: { baz: 'baz' } })
        .toEqual({ zoo: 'zoo', subObject: { baz: 'baz' } }); //
        PASS
});

test('toEqual with objects - fail', () => {
    expect({ zoo: 'zoo', subObject: { num: 0 } })
        .toEqual({ zoo: 'zoo', subObject: { baz: 'baz' } }); //
        FAIL
});

test('toEqual with arrays - pass', () => {
    expect([7, 19, 5]).toEqual([7, 19, 5]); // PASS
});

test('toEqual with arrays - fail', () => {
    expect([7, 19, 5]).toEqual([7, 19]); // FAIL
});
```

- **toContain()** - checks if the array or iterated object contains a value. The === operator is used for comparison. Replace the code in the IDE with the following code:

```
const arr = ['yellow', 'white', 'black'];

test('toContain with arrays', () => {
  expect(arr).toContain('black'); // PASS
});

test('toContain with sets', () => {
  expect(new Set(arr)).toContain('black'); // PASS
});

test('toContain with strings', () =>{
  expect('yellow, white, banana').toContain('black'); // FAIL
});
```

- **toContainEqual()** - checks if an array contains an element with the expected structure. Replace the code in the IDE with the following code:

```
test('toContainEqual', () => {
  expect([{a: 1}, {b: 2}]).toContainEqual({a: 1}); // PASS
});
```

- **toHaveLength()** - checks if the `length` property of the object matches what is expected.

- **toBeNull()** - checks for equality to null.
- **toBeUndefined()** - checks for equality to undefined.
- **toBeTruthy()** - checks in a boolean context the value is `true`. That is, any values except `false`, `null`, `undefined`, `0`, `NaN` and empty strings are considered to be `true`.
- **toBeGreaterThan()** and **toBeGreaterThanOrEqual()** - the first method checks if the passed numeric value is greater than expected `>`, the second checks greater than or equal to expected `>=`.
- **toMatch()** - checks if the string matches a regular expression. Replace the code in the IDE with the following code:

```
test('toMatch', () => {
  expect('Black').toMatch(/Bl/); // PASS
});
```

- **not** - this property allows you to make checks for inequalities. Replace the code in the IDE with the following code:

```
test('not with booleans', () => {
  expect(true).not.toBe(false); // PASS
});

test('not with objects', () => {
  expect({ foo: 'bar' }).not.toEqual({}); // PASS
});

test('not with functions', () => {
  function funcWithoutError() {}
  expect(funcWithoutError).not.toThrow(); // PASS
});
```

Checking Exceptions

Throwing an Error

Assume you have the following function that reverses a string or throws an exception when the parameter passed to the function is not a string.

```
function reverseString(str) {
  if (typeof(str) !== 'string') {
    throw new Error('Parameter not a string');
  }

  let splitString = str.split('');
  let reverseArray = splitString.reverse();
  let reverseString = reverseArray.join('');

  return reverseString;
}

module.exports = reverseString;
```

To test that the function properly throws an error, we can use the `toThrow()` method. According to the Jest documentation, you must wrap the function call in another function. If not, Jest will not catch the error and the assertion statement will fail.

Copy and paste the code below into the IDE. **Note**, this is not a complete set of tests, we are only interested in checking the exception.

```
const reverseString = require('./throw');

test('Check for error', () => {
  expect(() => {
    reverseString(7).toThrow('Parameter not a string');
  })
});
```

Notice how the `expect` statement uses an arrow function as a wrapper around `reverseString` followed by the `toThrow()` method. Click the button below to run the test.

challenge

Try This Variation

Instead of using an arrow function, use a named function as a wrapper in the test.

```
const reverseString = require('./throw');

test('Check for error', () => {
  function checkNonString() {
    reverseString(7);
  }

  expect(checkNonString).toThrow();
});
```

The `checkNonString` function does not take any parameters. In its body, call `reverseSting` with a parameter that will throw an error. Call the wrapper function in the `expect` statement so that Jest can capture the exception. Run the test again.

Types of Errors

In addition to checking if a function throws an error, you can test what error is thrown. The `reverseSting` function throws a new error with the string `'Parameter not a string'`. You can check for this exact error.

```
const reverseString = require('./throw');

test('Check for error', () => {
  expect(() => {
    reverseString(7).toThrow(new Error('Parameter not a
        string'));
  })
});
```

challenge

Try These Variations

- Perform a substring match when checking the exception.

```
const reverseString = require('./throw');

test('Check for error', () => {
  expect(() => {
    reverseString(7).toThrow('not a string');
  })
});
```

- Perform a substring match with a regular expression.

```
const reverseString = require('./throw');

test('Check for error', () => {
  expect(() => {
    reverseString(7).toThrow(/not a string/);
  })
});
```

Both of these tests pass because `'not a string'` is a substring of `'Parameter not a string'`.

Circles Example

Let’s create a couple of functions that we can test. First, let’s create a simple module that contains two functions for working with circles. These functions calculate the area and circumference. **Note**, curly braces are used around `area` and `circumference` so that we can refer to them individually.

```
const area = (radius) => Math.PI * radius ** 2;
const circumference = (radius) => 2 * Math.PI * radius;

module.exports = { area, circumference };
```

Next, let’s add the tests. Click the link below to open the test file.

First import the `circles.js` file which contains the functions we want to test. Then declare a test that checks the `area` function. The first test case expects `area(5)` to return a value close to `78.54`. We also want to check that if we call `area` without an argument, it will return a value that is not a number (`NaN`).

```
const circle = require('./circles');

test('Circle area', () => {
    expect(circle.area(5)).toBeCloseTo(78.54);
    expect(circle.area()).toBeNaN();
});
```

Now add a second test that checks the `circumference` function. Notice how the `toBeCloseTo` method takes two arguments. The second indicates that we are checking the return value up to the first decimal place. Again, we want to verify that calling `circumference` without an argument returns a value that is not a number (`NaN`).

```
test('Circumference', () => {
    expect(circle.circumference(11)).toBeCloseTo(69.1, 1);
    expect(circle.circumference()).toBeNaN();
});
```

Run the test with the terminal command below.

```
npm run test circles.test.js
```

In this test we checked the results of two functions - `area` and `circumference`. We did this using the `toBeCloseTo()` method. Pass the method two numbers representing the expected value and the precision for the comparison. If you pass it only one number, Jest uses 0.005 for the precision. Use `.toBeCloseTo()` when working with floating point numbers. If not, a small difference in rounding can lead to a test failing. The `toBeCloseTo()` method avoids this issue.

Grouping Tests

Feedback and Test Cases

Sometimes a failing test case does not give you actionable feedback. To illustrate this, let's create the `validateValue` function that takes a number and returns `true` if it's between 0 and 100. If not, it returns `false`. Add the function to the IDE.

```
const validateValue = (value) => {
    if(value < 0 || value > 100) {
        return false;
    }
    return true;
}

module.exports = validateValue;
```

Click the link below to open the test file and add the test to the IDE. Notice how we are testing a correct value and two incorrect values at the boundaries. **Important**, there is a bug in our test. The last test case should be `false`.

```
const validateValue = require('./validateValue');

test('Validate Value', () => {
    expect(validateValue(50)).toBe(true);
    expect(validateValue(-1)).toBe(false);
    expect(validateValue(101)).toBe(true); // ERROR
})
```

Run the test, which should fail.

```
npm run test validateValue.test.js
```

If you look through the Jest output, you will see the text below. We know the `Validate Value` test failed, but we don't know which test causes the problem.

```
FAIL  ./validateValue.test.js
 ✕ Validate Value (5 ms)
```

Grouping Tests with `describe`

Jest allows you to group related tests together with the `describe` keyword and a string that describes the grouping of tests. Instead of having three test cases, put each case in its own test.

```
const validateValue = require('./validateValue');

describe('Validating Different Values', () => {
    test('Validate Value', () => {
        expect(validateValue(50)).toBe(true);
    });
    test('< Value', () => {
        expect(validateValue(-1)).toBe(false);
    });
    test('> Value', () => {
        expect(validateValue(101)).toBe(true); // ERROR
    });
});
```

Run the test once again.

```
npm run test validateValue.test.js
```

The test still fails, but Jest gives us much better output that allows us to address the failing test. We see all three tests grouped together, and we have a name for the failing test. Fixing errors are much easier when structuring your tests with `describe`.

```
FAIL  ./validateValue.test.js
 Validating Different Values
   ✓ Validate Value (2 ms)
   ✓ < Value
   ✕ > Value (3 ms)
```

Additional Actions

So far, our test scripts have only run tests. You find yourself needing to perform actions either before or after the tests run.

Actions Before

Start by creating the function `sameType`. It takes an array and returns `true` of all elements in the array are of the same data type. If not, it returns `false`.

```
function sameType(arr) {
  typeArr = arr.map(val => typeof(val))
  return new Set(typeArr).size == 1;
}

module.exports = sameType;
```

Use the link below to open the test file.

Use `describe` to create two tests. The first has three test cases, which all produce `true`. The second test also has three test cases, but these should produce `false`.

```
const sameType = require('./actions')

describe('Validate sameType',() => {
  test('Correct Values', () => {
    expect(sameType([1, 2, 3, 4])).toBe(true);
    expect(sameType(['1', '2', '3', '4'])).toBe(true);
    expect(sameType([false, true, false, true])).toBe(true);
  });

  test('Incorrect Values', () => {
    expect(sameType([1, '2', 3, 4])).toBe(false);
    expect(sameType(['1', '2', '3', 4])).toBe(false);
    expect(sameType([false, true, 'false', true])).toBe(false);
  });
});
```

If you want to perform an action before each test, use the `beforeEach` function. It takes a callback function as a parameter. In this case, we are printing a string. Similarly, the `beforeAll` function is called before all of the tests. Update the code in the IDE to make use of these two functions.

```
const sameType = require('./actions')

describe('Validate sameType',() => {
  beforeEach( () => {
    console.log('Running before each test.');
  });
  beforeAll( () => {
    console.log('Running before all tests.');
  });
  test('Correct Values', () => {
    expect(sameType([1, 2, 3, 4])).toBe(true);
    expect(sameType(['1', '2', '3', '4'])).toBe(true);
    expect(sameType([false, true, false, true])).toBe(true);
  });

  test('Incorrect Values', () => {
    expect(sameType([1, '2', 3, 4])).toBe(false);
    expect(sameType(['1', '2', '3', 4])).toBe(false);
    expect(sameType([false, true, 'false', true])).toBe(false);
  });
});
```

Run the test. Be sure to scroll through the output to see the text logged to the console.

```
npm run test actions.test.js
```

▼ **Did you notice?**

The `beforeEach` function is defined before the `beforeAll` function. However, if you look at the test output, you see that Jest called `beforeAll` first.

Actions After

Just as you used `beforeEach` and `beforeAll` to perform tasks prior to tests running, you can use `afterEach` and `afterAll` to perform tasks after tests run. Again, the order does not matter. You can define `afterAll` first, but Jest will not execute these actions until after all of the tests have run.

```
const sameType = require('./actions')

describe('Validate sameType',() => {
  beforeEach( () => {
    console.log('Running before each test.');
  });
  beforeAll( () => {
    console.log('Running before all tests.');
  });
  test('Correct Values', () => {
    expect(sameType([1, 2, 3, 4])).toBe(true);
    expect(sameType(['1', '2', '3', '4'])).toBe(true);
    expect(sameType([false, true, false, true])).toBe(true);
  });

  test('Incorrect Values', () => {
    expect(sameType([1, '2', 3, 4])).toBe(false);
    expect(sameType(['1', '2', '3', 4])).toBe(false);
    expect(sameType([false, true, 'false', true])).toBe(false);
  });
  afterEach( () => {
    console.log('Running after each test.');
  });
  afterAll( () => {
    console.log('Running after all tests.');
  });
});
```

Run the tests again. Jest will log the appropriate string after the tests have finished running.

```
npm run test actions.test.js
```