Mark Westerlund, Jared Moser

SWENG 431: Software Verification, Validation, and Testing

Dr. Wen-Li Wang

1 May 2022

# What is Jest?

Jest is a unit testing framework for JavaScript that is agnostic, meaning that it can be used with almost all JavaScript frameworks and libraries. This includes popular frameworks such as React, Angular, and Vue. Since Jest configuration is simple, it allows integration into JavaScript builders Babel and WebPack easily as well. This also means that it seamlessly integrates with TypeScript, which is a version of JavaScript that is strongly typed. The growth of popularity of Jest is recognizable with millions of downloads per month and being utilized in millions of public repositories on GitHub. With Jest being open source and backed by major technology companies, there is an immense community for support and documentation making learning and utilizing Jest very user-friendly.

But how does Jest work? Jest works by taking advantage of the way JavaScript treats functions as objects. Since Jest runs in parallel with the main code base, overwriting functions with mocked functions and mocked values does not affect the code written during development. When Jest loads a module or a class, it creates a new instance of this class which allows Jest to overwrite functions and values keeping the source code completely decoupled from test code. Having the source code decoupled from the test code keeps the code baseline cleaner, more visible, and more efficient. It also allows the testing framework to be easily changed on the fly if desired.

Jest’s powerful mocking allows its users to easily isolate units down to the function level. Included with the framework are set up and tear down functions to properly set up each test in a reliable way and also tear down when completed. These test conditions can easily contain issues themselves due to human error, however, Jest helps handle these issues by introducing beforeEach setup in a test suite. This method allows the user to isolate conditions unique to a single test for more efficient debugging on found faults. Once the tests are properly set up and the tests are run, Jest’s output is easy to understand by showing expected outputs along with actual outputs, allowing the user to quickly identify the problem with either the main code baseline or how the test was written itself.

# Pros and Cons

## Pros:

### Easy Configuration

Jest’s configuration is extremely simple considering there is not much to setup on the user’s part. Jest is easily set up by installing it with one of the popular JavaScript package managers, such as NPM or Yarn. Once it is installed, it can be easily utilized by running the appropriate commands from the command line. Jest’s configuration documentation is easy to understand as well, which includes documentation on how to get started with simple examples and continues into more complex examples for deeper needs. On top of the official documentation, there is also a large amount of user generated content on popular forum sites such as StackOverflow and in the specific Jest documentations.

Jest also ships with many popular frameworks and libraries. For example, Jest is the preferred testing framework for React and the framework’s project creation command line interface automatically sets up and configures Jest out of the box. Any file with .test.js is recognized as a testing file and is run when the test commands are run.

### Simple but Rich API

Jest’s API is simple and easy to understand, which allows the user to divide tests into testing suites and as well as individual tests. Using the describe() function, the user can label a set of tests with either a class or a function name. Describe is also able to be nested to allow suites to exist inside other suites. This makes testing output extremely readable. Each test suite contains setup, tests, and tear down. The tests are written using the test() or it() functions, which are copies of each other. Each test must contain one or more expect() statements. The expect statements are structured with an expected value and actual value.

### Simple Mocking of JavaScript Functions

Jest allows functions to be “spied on” meaning that the testing framework is able to know when a function is called. This is extremely important for classes and objects that require specific states to exist, such as React components. This allows functions to be entirely isolated without having to set up a stateful object, increasing coverage and speed of testing. Jest is able to do this using JavaScript reflection to replace functions on the fly inside the testing instance. Jest’s mocking also makes it easy to test integration between units in a code base. Using the functions .mockImplementation() and .mockReturnValue() allows the user control over mocked functions keeping the tested function completely isolated from other functions in the code base.

### Easy to Understand Errors/Faults

Jest’s test output is very easy to understand since when all tests are run, the files are listed with a PASS or FAIL flag. From there it is easy to isolate the failed tests by using the command line to find the faulty tests. When looking into a specific file, All test suites are listed with their tests, with a red ‘X’ for a fail and a green check for a pass. All faulty tests are then listed with expected value, highlighted in green, and actual value, with the differences highlighted in red. The test is shown with the line number of the failed expect statement plus a few lines surrounding it. This helps quickly find the faulty test.

## Cons:

### Unable to Mock Language Object Constructors

One major drawback of Jest is its inability to easily to mock native language objects. While this is not often needed, this drawback can make writing certain types of unit tests difficult. For example, it is difficult to spy on the creation of a native JavaScript array when using the new keyword. This can leave holes testing coverage.

### Mocking Asynchronous Functions

While Jest works extremely well with pure JavaScript functions, meaning the function takes in parameters and returns a single object, using asynchronous functions can be difficult and not straightforward despite the API and documentation. Jest provides APIs for resolving and rejecting Promises to get asynchronous values but requires a different API for testing whether a Promise object has been returned. It is often not straight forward to know when to use which.

### Has Trouble Mocking Functions with Recursion

Jest has trouble mocking return vales from functions that are called inside a loop. There are several ways to handle these types of integrations, but none are straight forward. Jest provides a function mockReturnValueOnce() that allows the function to be mocked with a specific value one time, but this requires a new return value for each iteration. The mockImplementation() function as able to accept a callback function as a parameter, but can lead to faults in the tests themselves.

### Strictly handles Unit Testing

Jest specifically does unit testing only, but while this is desired, there is no way to test integration with a web based API. Since JavaScript is mainly used inside of a browser, connecting with web based APIs is integral to modern JavaScript development. Other testing frameworks are required to test these types of integration.

# Functional Methods

## Globals

Jest provides several global functions that help set up and tear down tests. These functions also help separate out tests and testing suites. Testing suites and tests themselves can each be considered a testing block. In general, each of these testing blocks takes another function as a parameter that is used to run all testing blocks nested inside of the function.

* **Test**

The test function is the integral function of Jest. The test function takes in two parameters, a name and a function. The function object that is passed into the test contains all of the actual test code, using expect statements. Jest also provides an alias for the test function called it which some users prefer as it helps with writing test descriptions.

test('name of the test', () => {  
 // testing statements  
});

* **Describe**

The describe function is used to separate out and group tests that are related. Describe also allows other describe blocks to be nested inside of other blocks, which is useful when testing an entire class. The describe function takes in two parameters, a name and a function object. The function object is where all nested describe blocks and tests are written. Describe blocks are not required for Jest to work, but they provide an easy way to group tests which makes them more human readable. Describe functions must contain at least one test or Jest will error when run.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 test('name of the test', () => {  
 // testing statements  
 });  
 });  
});

* **BeforeAll**

The beforeAll() function is run once before every testing block. This allows the set up of program states that are required to properly test. The before all function takes in another function that will run when the test is ready.

describe(myClass.name, () => {  
 let state;  
 beforeAll(() => {  
 state = {my: 'state'}  
 });  
  
 test('name of the test', () => {  
 // state is already initialized here  
 // testing statements  
 });  
});

* **BeforeEach**

The beforeEach() function is run before every test inside a block is run. Every time Jest encounters a new test or describe block, the beforeEach function in that scope is run. BeforeEach also takes in a function that is executed when beforeEach is called.

describe(myClass.name, () => {  
 let state;  
 beforeEach(() => {  
 state = {my: 'state'}  
 });  
  
 test('name of the test', () => {  
 // state is already initialized here  
 // testing statements  
 });  
  
 test('name of the test', () => {  
 // state is reinitialized here  
 // testing statements  
 });  
});

* **AfterEach**

The afterEach() function is called after every testing block is run. Every time Jest finishes executing a test or describe block, the afterEach function is called. This is useful for restoring mocked functions to their original objects.

describe(myClass.name, () => {  
 let state;  
 afterEach(() => {  
 myFunction.restore();  
 });  
  
 test('name of the test', () => {  
 // testing statements  
 });  
 // after each is called here  
  
 test('name of the test', () => {  
 // testing statements  
 });  
 // after each is also called here  
});

* **AfterAll**

The afterAll() function is called after all tests are completed within a block. This is useful for restoring values after all tests inside a block are completed.

describe(myClass.name, () => {  
 let state;  
 afterAll(() => {  
 myFunction.restore();  
 });  
  
 test('name of the test', () => {  
 // testing statements  
 });  
  
 test('name of the test', () => {  
 // testing statements  
 });  
 // after all is called here  
});

### Expects

When running unit tests, expected values are usually tested against actual returned values. Jest provides the Expect API with a wide set of functions to facilitate testing these values. The expect function is run inside of testing blocks. Expect functions are typically run with a “matcher” function that will be checked against the expected value passed into the the expect. There are many matcher functions that help with handling common situations encountered when testing functions. These include matchers that help handling numbers, arrays, and objects. There are also matchers that help with handling asynchronous functions. These are a few common features used while writing unit tests.

* **Expect(value)**

Every test will contain an expect() call. This is where the actual value testing happens. The general pattern for using is expect(actual).matcher(expected) where actual is the actual value and expected is the expected value. When Jest encounters a failed test, it isolates the first expect statement failure for that test in the response to help the tester quickly find the faulty test.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 it('returns the hello world response', () => {  
 expect(myClass.helloWorld()).toBe('Hello, world!');  
 });  
 });  
});

* **.toEqual()**

When writing unit tests, the tested unit often should return an exact value. The .toEqual() matcher takes in the expected value and tests whether or not the actual value is exactly equal to the expected value. The .toEqual matcher is able to take in any data type including number, string, Array, and Object. Importantly, the .toEqual recursively compares all object properties to check for deep equality.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 test('1 + 2 = 3', () => {  
 expect(1 + 2).toEqual(3);  
 });  
  
 test('1 + 2 = 3', () => {  
 expect(1 + 2).toEqual('3'); // This will fail, because the expected value is a string  
 });  
  
 test('deep equality', () => {  
 const myObject = {my: 'object', count: 1};  
 expect(myObject).toEqual({my: 'object', count: 1}); // This will pass because of deep equality  
 });  
 });  
});

* **.not**

All matchers are able to be negated py adding the .not. modifier. This is extremely useful when a function should return a value in some instances but never return that value otherwise.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 test('1 + 2 = 3', () => {  
 expect(1 + 2).not.toEqual(5);  
 });  
  
 test('1 + 2 = 3', () => {  
 expect(1 + 2).not.toEqual(3); // This will fail, because the expected value is a string  
 });  
 });  
});

* **Expect.any()**

Sometimes when a function is tested, the actual values are not important, but the class of the object returned is. Jest provides a modifier to the expect function that will return a new object of the constructor passed in. This is useful when testing the creation of an object inside a function and the value is returned.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 test('the function returns a new object', () => {  
 expect(function1()).toEqual(Expect.any(Object));   
 // This will be true for any response for function1 that is an Object  
 // but will fail if its response is a primitive type like number  
 });  
 });  
});

* **.toBeUndefined()**

Sometimes a value should be undefined, null, or NaN (not a number). Jest provides matchers that explicitly test for these values.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 test('the function returns undefined', () => {  
 expect(function1()).toBeUndefined());   
 });  
 });  
});

* **.toHaveBeenCalled()**

Many functions contain calls to other functions. Jest provides a matcher that works in conjunction with the Mock api that allows the testing of function calls. The matcher .toHaveBeenCalled() tests whether the function has been called at least once. Other matchers include .toHaveBeenCalledTimes(), toHaveBeenCalledWith(arg1, arg2, ...) and .toHaveBeenCalledLastWith(...).

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 beforeEach(() => {  
 myClass.function2 = jest.fn();  
 })  
 test('the function returns a new object', () => {  
 myClass.function1()  
 expect(myClass.function2).toHaveBeenCalled(); // Passes when function1 calls function2  
 });  
 });  
});

### Mock

Jest’s Mock API is its most powerful feature. This API allows Jest to completely isolate units and allows the testing of integration between units. Mock functions, also know as “spies”, are able to intercept calls between functions and change their implementation. This lets each unit test to have the testing parameters entirely determined for each test by the tester. It also allows tests to know whether or not a call has happened from inside another function. This lets a tester have complete control and coverage over their unit testing. Jest is able to mock functions and entire modules.

* **SpyOn**

The spyOn function takes in an object and a function name as its parameters, and replaces the function as desired. Jest will error if the function name passed in does not exist on that object, which is useful for testing existence of functions.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 beforeEach(() => {  
 jest.spyOn(myClass, 'function2').mockImplementation();  
 })  
 test('the function returns a new object', () => {  
 myClass.function1()  
 expect(myClass.function2).toHaveBeenCalled(); // Passes when function1 calls function2  
 });  
 });  
});

* **Jest.fn()**

Jest also provides a way to mock a function if it belongs to a nested object. For example, if an function is passed into a react prop, the function can be mocked by reassigning it to a jest.fn(). This is also useful for spying on functions attached to the global object in javascript, such as fetch().

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 let theUrl;  
 beforeEach(() => {  
 theUrl = '/my/url/';  
 window.fetch = jest.fn().mockImplementation();  
 })  
 test('the function returns a new object', () => {  
 myClass.function1()  
 expect(window.fetch).toHaveBeenCalledWith(theUrl); // Passes when function1 calls fetch with '/my/url/'  
 });  
 });  
});

* **MockImplementation()**

Jest mock functions are able to change their implementation on the fly for each testing block. This is useful when a function return value is not important inside another function, but the call itself is important.

describe(myClass.name, () => {  
 describe(myClass.function1.name, () => {  
 beforeEach(() => {  
 myClass.function3 = jest.fn().mockImplementation((i) => i.id);  
 })  
 test('the function returns a new object', () => {  
 myClass.function1()  
 expect(myClass.function2).toHaveBeenCalledWith(id);  
 // In this instance, function2 is called with function3s response like function2(function3({id: 1}));  
 // Function 3 is mocked to return the id of the object called to, which allows the testing of controlled input to function2  
 });  
 });  
});

* **MockReturnValue()**

Jest also allows the mocking of a return value from a function. This allows the complete encapsulation of one unit test from another function.

describe(myClass.function1.name, () => {  
 beforeEach(() => {  
 jest.spyOn(myClass, 'function2').mockReturnValue('returned');  
 })  
 test('the function returns a new object', () => {  
 expect(myClass.function2()).toEqual('returned'); // This test will always pass  
 });  
});

# Functional Features

## Unit tests

## Jest CLI

The Jest CLI (Command Line Interface) is where the testing output is displayed. Once the CLI is up and running, it automatically watches the test files set up in the project directory and runs again whenever a file is saved. While the CLI is waiting, commands can be added to run certain tests:

Watch Usage  
 › Press f to run only failed tests.  
 › Press o to only run tests related to changed files.  
 › Press q to quit watch mode.  
 › Press i to run failing tests interactively.  
 › Press p to filter by a filename regex pattern.  
 › Press t to filter by a test name regex pattern.  
 › Press Enter to trigger a test run.

### Run failed tests

Using the f option, Jest automatically runs the tests that have failed and displays the details:

FAIL src/components/TodoInput/TodoInput.test.tsx  
 TodoInput  
 \_textChange  
 ✕ sets the state when the text input changes (FAILS ON PURPOSE) (5 ms)  
 ✓ sets the state when the text input changes  
 \_isEnabled  
 ✓ returns true when the input has a value (1 ms)  
 ✓ returns false when the input dose not have a value  
 \_saveTodo  
 ✓ calls save todo  
 ✓ resets the state to empty (1 ms)  
 ✓ does not call the functions when there is no input value  
  
 ● TodoInput › \_textChange › sets the state when the text input changes (FAILS ON PURPOSE)  
  
 expect(jest.fn()).toHaveBeenCalledWith(...expected)  
  
 - Expected  
 + Received  
  
 Object {  
 - "todo": "theValue",  
 + "todoValue": "theValue",  
 },  
  
 Number of calls: 1  
  
 22 | it('sets the state when the text input changes (FAILS ON PURPOSE)', () => {  
 23 | subject.\_textChange(event);  
 > 24 | expect(subject.setState).toHaveBeenCalledWith({todo: value});  
 | ^  
 25 | });  
 26 |  
 27 | it('sets the state when the text input changes', () => {  
  
 at Object.<anonymous> (src/components/TodoInput/TodoInput.test.tsx:24:32)  
  
Test Suites: 1 failed, 1 total  
Tests: 1 failed, 6 passed, 7 total  
Snapshots: 0 total  
Time: 1.746 s, estimated 2 s  
Ran all test suites.

Reviewing the test output above from top to bottom:

* The test suites that have failed are labeled within their describe blocks along with the test name. The failed test is marked with a ‘✕’.
* A detailed description for each failed test is shown. It shows specifically what test failed, and the expected and received values.
* After the expected and received values, Jest displays the line number for which test failed, along with some context to allow the tester to understand what and where the error occurred in the test.
* Finally, after this input, there is a summary listing details about all tests that were run.

### All tests

Making output extremely readable to the tester, when running all tests, Jest lists out each test file that has been run. Also included is a PASS or FAIL label, as well as, the total run details:

PASS src/components/TodoList/TodoList.test.tsx  
 PASS src/App.test.tsx  
 FAIL src/components/TodoInput/TodoInput.test.tsx

Test Suites: 1 failed, 2 passed, 3 total  
Tests: 1 failed, 13 passed, 14 total  
Snapshots: 0 total  
Time: 1.775 s, estimated 2 s  
Ran all test suites.

### File pattern

Jest also allows the tester to enter a regex pattern to run a specific file or test:

Pattern Mode Usage  
 › Press Esc to exit pattern mode.  
 › Press Enter to filter by a filenames regex pattern.  
  
 pattern › TodoI  
  
 Pattern matches 1 file  
 › src/components/TodoInput/TodoInput.test.tsx

Once a file is selected to run, the output is specific to that file

PASS src/components/TodoList/TodoList.test.tsx  
 TodoList  
 \_getTodoItem  
 ✓ returns a list item (3 ms)  
 ✓ returns a list item when the index is 0 (1 ms)  
 ✓ returns undefined if the todo is undefined/null (1 ms)  
 ✓ returns undefined if the index is undefined/null (1 ms)  
 \_getTodoItems  
 ✓ gets todo items (3 ms)  
 ✓ returns the todos (1 ms)  
  
Test Suites: 1 passed, 1 total  
Tests: 6 passed, 6 total  
Snapshots: 0 total  
Time: 0.487 s, estimated 1 s  
Ran all test suites matching /src\/components\/TodoList\/TodoList\.test\.tsx/i.  
  
Watch Usage: Press w to show more.

### Conclusion

Jest is a very popular and useful unit testing tool that is easily integrated into the most popular frameworks. Knowledge of a popular framework such as React makes integrating Jest that much more feasible. By testing web components away from the main code baseline, testers can pinpoint specific functions within specific components while utilizing Jest API features to create a user experience that is locked down from bugs.

### References

[1] “Getting Started,” Jest Docs, 25-Apr-2022. [Online]. Available: <https://jestjs.io/docs/getting-started> .

[2] “Understanding Jest,” *DEV.to*. [Online]. Available: <https://dev.to/ekqt/understanding-jest-2ifm>.

[3] “Understanding Jest Mocks,” *Medium*, 08-Mar-2018. [Online]. Available: <https://medium.com/@rickhanlonii/understanding-jest-mocks-f0046c68e53c>.