Testing and Debugging in JavaScript

Adrián García Rodríguez, Rubén Díaz Marrero,
Ramón Izquierdo Izquierdo

TABLE OF CONTENTS

Introduction to Testing

Key Benefits of Testing

Bug Prevention

Identifies issues early, reducing the time and cost of fixing them later.

Improved Code Quality

Ensures reliability, maintainability, and scalability.

Supports Refactoring

Allows developers to modify code without fear of breaking functionality.

Testing Methodologies

- **Manual Testing**
- **Automated Testing**

Common Testing Strategies

- Unit Testing (UT)
- Integration Testing
- End-to-End (E2E) Testing
- Test-Driven Development
- Behavior-Driven Development

Jest Framework

What is a Framework?

Structured environment that provides code, tools and guidelines

Key Features:

- **Has a predefined structure**
- **Inversion of control**
- **Reusable components**
- **Encapsulation**

Helps in building web servers easily

What is a Test Framework?

Specialized framework designed to help developers in testing code.

Key Features:

- **Test runner (.spec.js/.test.js)**
- **Result assertions**
- **Mocking and spying**
- **Code coverage**

Test Framework Example

Implement function that is gonna be tested and export it

Import the function to test and execute the test code

main.py

```
function add(number1, number2) {
    return number1 + number2;
}

module.exports = add;
```

math.js | math.test.js

```
const add = require('./math'); // Import function

test('adds 2 + 3 to equal 5', () => {
  expect(add(2, 3)).toBe(5); // Jest assertion
});
```

Main Testing Frameworks

Main JavaScript Testing Frameworks

Framework	Purpose	Features
Jest	General testing	snapshot testing, code coverage..
Mocha	Flexible test runner	Works with assertion libraries like Chai
Chai	Assertion library	Provides syntax like assert(), expect()..
Cypress	End-to-End test	Simulates user interactions in a browse
Playwright	Browser test	Test web apps across different browsers

Which one to choose?

MochaJs

Is a flexible test runner that allows you to write test in JavaScript

Main difference between Jest and Mocha

- **Doesn't include assertions**
- **Doesn't include mocking tools**
- **Slower (serial execution)**
- **Depends on other frameworks**

In order to fix the lack of functionality, MochaJs works normally aside with Chai & Sinon which are both type of Test Framework

MochaJs & Sinon Example

Examples implementation of tests using Mocha and Sinon, for spying and mocking function.

logger.js

```
function logMessage(message) {
  console.log(message);
}
module.exports = logMessage;
```

logger.test.js

```
const sinon = require('sinon');
const logMessage = require('./logger');

describe('Sinon Spy Example', function () {
  it('should call logMessage with correct argument', function () {
    const spy = sinon.spy(console, 'log');

    logMessage('Test Message');

    sinon.assert.calledWith(spy, 'Test Message');

    spy.restore(); // Restaurar función original
  });
});
```

What is the Jest Test Framework?

- **Is a JavaScript testing framework**
- **Primarily designed for React based apps**
- **Provides complete testing solution**

other frameworks don´t include all functionality

Jest Example

math.js | math.test.js

```
1  const add = require('./math'); // Import function
2
3  test('adds 2 + 3 to equal 5', () => {
4    expect(add(2, 3)).toBe(5); // Jest assertion
5  });
```

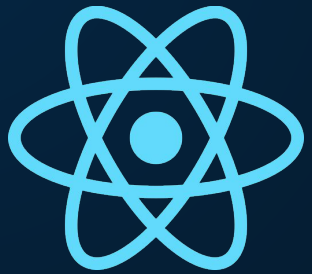
Jest Key Features

- Easy set up, zero configuration needed
- Fast execution
- Built-in Mocks and Spies
- Snapshot testing
- Runs in Node.js and browsers
- Code Coverage
- Asynchronous Testing
- Fake Timers
- Works with TypeScript and ESModules
- Watch mode

Why Jest?

Facebook created Jest for React

Popularity led to the use of Jest for Javascript



Why Unit Testing?

Very fast to execute

Easy to setup

Not requiring any elaborate configuration

Very precise feedback

Installing and Configuring Jest

Installing globally

```
$ npm install -g jest
```

```
$ jest --version
```

Checking correct installation

Installing and Configuring Jest

```
$ npm init -y
```

Creating project

```
{
  "name": "first-jest-tests",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
}
```

Installing and Configuring Jest

Adding Jest to the project

```
$ npm install --save-dev jest
```

```
{
  "name": "first-jest-tests",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "jest": "^29.7.0"
  }
}
```

The ‘test’ script will run Jest when it is used.

Installing and Configuring Jest

```
function fizzBuzz(numericSequence) {
  let resultSequence = [];
  for (const number of numericSequence) {
    if (number % 15 === 0) {
      resultSequence.push('fizzbuzz');
    } else if (number % 3 === 0) {
      resultSequence.push('fizz');
    } else if (number % 5 === 0) {
      resultSequence.push('buzz');
    } else {
      resultSequence.push(number);
    }
  }
  return resultSequence.join(', ');
}

module.exports = fizzBuzz;
```

FizzBuzz Problem:

FizzBuzz is a common coding task given during interviews that tasks candidates to write a solution that prints integers one-to-N.

Any integers divisible by three is labeled as “Fizz,” integers divisible by five as “Buzz” and integers divisible by both three and five as “FizzBuzz.”

Installing and Configuring Jest

```
const fizzBuzz = require('./fizz-buzz-interview');

describe("FizzBuzz", () => {
  test('[3] should result in "fizz"', () => {
    expect(fizzBuzz([3])).toBe('fizz');
  });

  test('[5] should result in "buzz"', () => {
    expect(fizzBuzz([5])).toBe('buzz');
  });

  test('[15] should result in "fizzbuzz"', () => {
    expect(fizzBuzz([15])).toBe('fizzbuzz');
  });

  test('[1,2,3] should result in "1, 2, fizz"', () => {
    expect(fizzBuzz([1, 2, 3])).toBe('1, 2, fizz');
  });
});
```

```
$ npm run test
```

Now we ready to run our first test. It's that easy!

Installing and Configuring Jest

```
module.exports = {
  verbose: true,
  projects: ['<rootDir>'],
  testMatch: [
    '**/__tests__/**/*.[jt]s?(x)',
    '**/test/**/*.[jt]s?(x)',
    '**/?(*.)+(spec|test).[jt]s?(x)',
  ],
  testPathIgnorePatterns: [
    '/(?:production_)?node_modules/',
    '.d.ts$',
    '<rootDir>/test/fixtures',
    '<rootDir>/test/helpers',
    '__mocks__',
  ],
  transform: {
    '^.+\\.[jt]sx?$': 'babel-jest',
```

The jest.config.js file

Configures Jest by customizing how tests are executed in a project.

Installing and Configuring Jest

```
module.exports = {
  verbose: true,
};
```

verbose:

Displays additional details about each test run.

testMatch:

Defines the filename patterns that Jest will use as tests.

```
module.exports = {
  testMatch: ['**/__tests__/**/*.test.js', '**/?(*.)+(spec|test).js'],
  testPathIgnorePatterns: ['/node_modules/', '/dist/'],
};
```

testPathIgnorePatterns:

Exclude directories such as node_modules/ and dist/ to avoid unnecessary testing.

Installing and Configuring Jest

```
module.exports = {
  transform: {
    '^.+\\.jsx?$': 'babel-jest',
  },
};
```

transform:

Use Babel (babel-jest) to convert modern code before executing it.

```
module.exports = {
  testEnvironment: 'node',
};
```

testEnvironment:

Define the execution environment (node for backend or jsdom for frontend).

Installing and Configuring Jest

```
module.exports = {
  moduleNameMapper: {
    '^@components/(.*)$': '<rootDir>/src/components/$1',
  },
};
```

moduleNameMapper:

Allows you to define aliases for module paths, useful when using @ in imports.

Matchers in Jest

1. **Matchers for Equality**
2. **Truthiness Matchers**
3. **Numeric Matchers**
4. **String Matchers**
5. **Array and List Matchers**
6. **Exception Matchers**

Matchers for Equality

- toBe(value): Checks for strict equality (===).
- toEqual(object): Used for checking objects or arrays.

```
test('los objetos pueden ser iguales', () => {
  const user = { name: 'Rubén' };
  expect(user).toEqual({ name: 'Rubén' }); // toBe fallaría
});
```

Truthiness Matchers

- toBeNull(): Checks if the value is null.
- toBeUndefined(): Checks if the value is undefined.
- toBeTruthy() / toBeFalsy(): Evaluates whether a value is considered true or false.

```
test('comprobando valores truthy y falsy', () => {
  expect(0).toBeFalsy();
  expect('hola').toBeTruthy();
});
```

Numeric Matchers

- toBeGreaterThan(n)
- toBeLessThan(n),
- toBeGreaterThanOrEqual(n)
- toBeLessThanOrEqual(n).

```
test('10 es mayor que 5', () => {
  expect(10).toBeGreaterThan(5);
});
```

String Matchers

- toMatch(regex): Checks if a string matches a regular expression.

```
test('el texto contiene Jest', () => {
  expect('Aprendiendo Jest es divertido').toMatch(/Jest/);
});
```

Array and List Matchers

- toContain(item): Checks if an array contains a specific item.

```
test('el array incluye el número 3', () => {
  expect([1, 2, 3]).toContain(3);
});
```

Exception Matchers

- toThrow(): Ensures that a function throws an error.

```
function throwError() {
  throw new Error('Unexpected error');
}

test('function throws an error', () => {
  expect(() => throwError()).toThrow('Unexpected error');
});
```

Some other matchers

- toHaveLength(n)
- toBeInstanceOf(Class)
- toMatchObject(obj)
- toMatchSnapshot()
- toStrictEqual(obj)
- toHaveReturnedWith(value)
- toSatisfy(fn)
- toHaveReturnedTimes(n)

Mock

```
function fetchData() {
  return 'Real API data';
}

module.exports = fetchData;
```

A mock completely replaces a function with a fake version that you can control.

Useful for:

- Simulate functions
- Returning specific values
- Spy on the way they are called

```
const fetchData = require('./api');

jest.mock('./api');

test('fetchData should return false data', () => {
  fetchData.mockReturnValue('False data');
  expect(fetchData()).toBe('False data');
});
```

Spy

```
function sum(number1, number2) {
  return number1 + number2;
}

module.exports = { sum };
```

A spy observes a real function without modifying it, but records how many times it was called, with what arguments, etc.

```
const  sumModule = require('./sum');

test('sum should be called correctly', () => {
  const spy = jest.spyOn(sumModule, 'sum');

  sumModule.sum(2, 3);
  sumModule.sum(5, 7);

  expect(spy).toHaveBeenCalledTimes(2);
  expect(spy).toHaveBeenCalledWith(2, 3);
  expect(spy).toHaveBeenCalledWith(5, 7);
  expect(sumModule.sum(3, 6)).toBe(9);

  spy.mockRestore();
});
```

Test Structure

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

describe('Calculator function tests', () => {
  test('Adds 2 + 3 correctly', () => {
    expect(calculator('sum', 2, 3)).toBe(5);
  });

  test('Subtracts 10 - 4 correctly', () => {
    expect(calculator('subtract', 10, 4)).toBe(6);
  });

  test('Multiplies 3 * 3 correctly', () => {
    expect(calculator('multiply', 3, 3)).toBe(9);
  });
```

describe:

Describe block is used for organizing test cases in logical groups of tests.

test:

Starts a new test case definition. *it* keyword is an alias for the test keyword.

xtest:

Disabled version of *test*. It is used to temporarily ignore a test without removing it from the code.

expect:

Main assertion function in Jest.

Snapshot Testing

What is Snapshot Testing?

<\/>

Technique used for software tests, especially in UI development.

Why is Snapshot Testing used for?

<\/>

Snapshots tests is a very useful tool in order to make sure UI (User Interface) doesn't change in an unexpected way along time

Usually used in Jest to check Reacts components or other frameworks.

What is Snapshot Testing purpose?

It ensures the changes in code doesn't affect the appearances or behavior of the components

Example:

button.js

```jsx
import React from 'react';

const Button = ({ label }) => {
  return <button>{label}</button>;
};

export default Button;
```

button.test.js

```jsx
import React from 'react';
import { render } from '@testing-library/react';
import Button from './Button';

describe('Button component', () => {
  it('should match the snapshot', () => {
    const { asFragment } = render(<Button label="Click me" />);
    expect(asFragment()).toMatchSnapshot();
  });
});
```

Snapshot Testing process

1. Initial Snapshot Creation

Snapshot testing captures the current output/state of component, and it is saved in a file in a folder called __snapshots__

2. Running tests

Every time tests are executed, the testing framework compares the output with the previous saved snapshot. Matcher: toMatchSnapshot()

2. Test result

if the snapshot matches, the tests are passed
if not, the tests fails and indicates the UI component changed.

Case Study

```
function createButton(label) {
  const button = document.createElement('button');
  button.textContent = label;
  // button.classList.add("btn-primary");
  return button;
}

module.exports = createButton;
```

```
$ npm test -- -u
```

Updating Snapshot

```
const createButton = require('./button');

test('createButton should return a button with correct label', () => {
  const button = createButton('Click Me');
  expect(button.outerHTML).toMatchSnapshot();
});
```

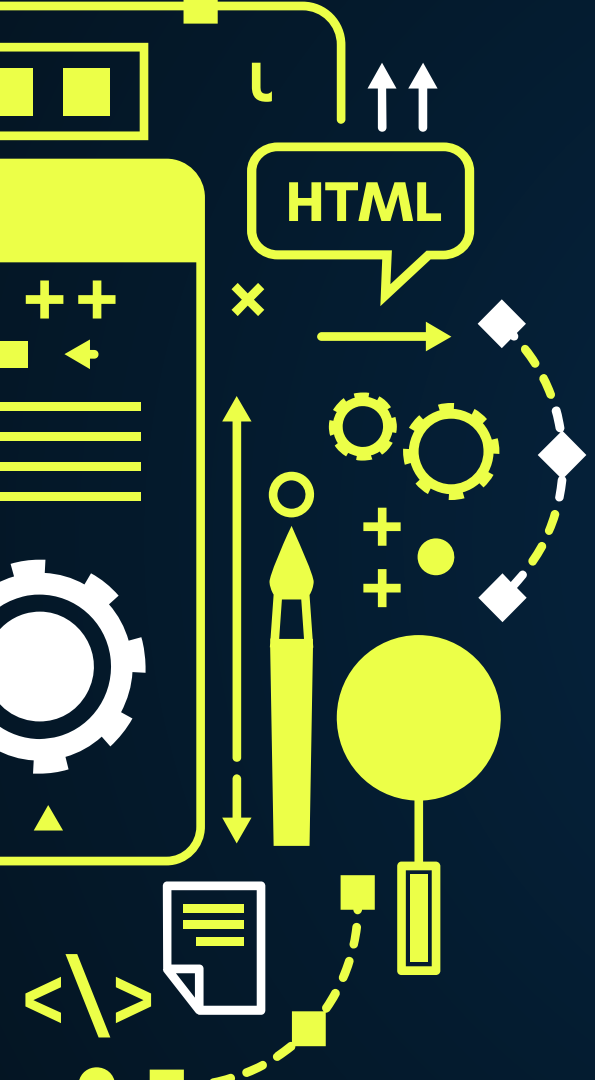
Debugging

what is debug?

The process of finding and fixing errors in the code you are analyzing. Identifying the errors, knowing what causes them and fix them in order to get a correct execution.

Debugging steps:

- **Reproduce the error**
- **Identify the cause**
- **Fix the error**
- **Try again execution**



THANKS!

Does anyone have any question?

Adrián García Rodríguez - adrian.garcia.rodriguez.36@gmail.com
Rubén Díaz Marrero ruben.diaz.17@ull.edu.es
Ramón Izquierdo Izquierdo alu0101480824@ull.edu.es

RESOURCES

- Jest
- Installation of NVM
- Jest Testing: A Helpful, Introductory Tutorial
- Jest configuration (jest.config.js)
- Test Unitarios en JavaScript
- Snapshot
- Node.js tutorial in VSC
- Node.js debugging in VSC