Unit testing framework for JavaScript – JEST

Unit testing is a software testing approach where individual units or components of a program are tested independently to ensure they function correctly. It involves testing each unit in isolation to validate its behavior, often done through automated scripts, to detect bugs early in the development process.

To implement our project KontaNiben, as we are using Node.js we will use JEST for unit testing.

JEST is a delightful JavaScript Testing Framework with a focus on simplicity, It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more. Jest provides a simple and efficient way to write tests, with features like built-in assertions, mocking capabilities, snapshot testing, and parallel test execution.

Install Jest using your favorite package manager:

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

Let's get started by writing a test for a hypothetical function that adds two numbers. First, create a sum.js file:

```
function sum(a, b) {
  return a + b;
}
module.exports = sum;
```

Then, create a file named sum.test.js. This will contain our actual test:

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

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Add the following section to your `package.json`:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Finally, run `yarn test` or `npm test` and Jest will print this message:

```
PASS  ./sum.test.js
✓ adds 1 + 2 to equal 3 (5ms)
```

You just successfully wrote your first test using Jest!

This test used `expect` and `toBe` to test that two values were exactly identical.

Running from command line

You can run Jest directly from the CLI (if it's globally available in your `PATH`, e.g. by `yarn global add jest` or `npm install jest --global`) with a variety of useful options.

Here's how to run Jest on files matching `my-test`, using `config.json` as a configuration file and display a native OS notification after the run:

```
jest my-test --notify --config=config.json
```

Additional Configuration

Generate a basic configuration file

Based on your project, Jest will ask you a few questions and will create a basic configuration file with a short description for each option:

```
npm init jest@latest
```

Using Matchers

Jest uses "matchers" to let you test values in different ways. This document will introduce some commonly used matchers. For the full list, see the expect API doc.

- Common Matchers

The simplest way to test a value is with exact equality.

```
test('two plus two is four', () => {
  expect(2 + 2).toBe(4);
});
```

In this code, expect(2 + 2) returns an "expectation" object. You typically won't do much with these expectation objects except call matchers on them. In this code, .toBe(4) is the matcher. When Jest runs, it tracks all the failing matchers so that it can print out nice error messages for you.

toBe uses Object.is to test exact equality. If you want to check the value of an object, use toEqual:

```
test('object assignment', () => {
  const data = {one: 1};
  data['two'] = 2;
  expect(data).toEqual({one: 1, two: 2});
});
```

toEqual recursively checks every field of an object or array.

toEqual ignores object keys with undefined properties, undefined array items, array sparseness, or object type mismatch. To take these into account use toStrictEqual instead.

You can also test for the opposite of a matcher using not:

```
test('adding positive numbers is not zero', () => {
  for (let a = 1; a < 10; a++) {
    for (let b = 1; b < 10; b++) {
      expect(a + b).not.toBe(0);
    }
  }
});
```

- Truthiness

In tests, you sometimes need to distinguish between undefined, null, and false, but you sometimes do not want to treat these differently. Jest contains helpers that let you be explicit about what you want.

- toBeNull matches only null
- toBeUndefined matches only undefined
- toBeDefined is the opposite of toBeUndefined
- toBeTruthy matches anything that an if statement treats as true
- toBeFalsy matches anything that an if statement treats as false

For example:

```
test('null', () => {
  const n = null;
  expect(n).toBeNull();
  expect(n).toBeDefined();
  expect(n).not.toBeUndefined();
  expect(n).not.toBeTruthy();
  expect(n).toBeFalsy();
});

test('zero', () => {
  const z = 0;
  expect(z).not.toBeNull();
  expect(z).toBeDefined();
  expect(z).not.toBeUndefined();
  expect(z).not.toBeTruthy();
  expect(z).toBeFalsy();
});
```

You should use the matcher that most precisely corresponds to what you want your code to be doing.

- Numbers

Most ways of comparing numbers have matcher equivalents.

```
test('two plus two', () => {
  const value = 2 + 2;
  expect(value).toBeGreaterThan(3);
  expect(value).toBeGreaterThanOrEqual(3.5);
  expect(value).toBeLessThan(5);
  expect(value).toBeLessThanOrEqual(4.5);

  // toBe and toEqual are equivalent for numbers
  expect(value).toBe(4);
  expect(value).toEqual(4);
});
```

For floating point equality, use toBeCloseTo instead of toEqual, because you don't want a test to depend on a tiny rounding error.

```
test('adding floating point numbers', () => {
  const value = 0.1 + 0.2;
  //expect(value).toBe(0.3);           This won't work because of rounding error
  expect(value).toBeCloseTo(0.3); // This works.
});
```

- Strings

You can check strings against regular expressions with toMatch:

```
test('there is no I in team', () => {
  expect('team').not.toMatch(/I/);
});

test('but there is a "stop" in Christoph', () => {
  expect('Christoph').toMatch(/stop/);
});
```

Arrays and iterables

You can check if an array or iterable contains a particular item using toContain:

```
const shoppingList = [
  'diapers',
  'kleenex',
  'trash bags',
  'paper towels',
  'milk',
];

test('the shopping list has milk on it', () => {
  expect(shoppingList).toContain('milk');
  expect(new Set(shoppingList)).toContain('milk');
});
```

- Exceptions

If you want to test whether a particular function throws an error when it's called, use toThrow.

```
function compileAndroidCode() {
  throw new Error('you are using the wrong JDK!');
}

test('compiling android goes as expected', () => {
  expect(() => compileAndroidCode()).toThrow();
  expect(() => compileAndroidCode()).toThrow(Error);

  // You can also use a string that must be contained in the error message or a regexp
  expect(() => compileAndroidCode()).toThrow('you are using the wrong JDK');
  expect(() => compileAndroidCode()).toThrow(/JDK/);

  // Or you can match an exact error message using a regexp like below
  expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK$/); // Test fails
  expect(() => compileAndroidCode()).toThrow(/^you are using the wrong JDK!$/); // Test pass
});
```

Testing Asynchronous Code

It's common in JavaScript for code to run asynchronously. When you have code that runs asynchronously, Jest needs to know when the code it is testing has completed, before it can move on to another test. Jest has several ways to handle this.

- Promises

Return a promise from your test, and Jest will wait for that promise to resolve. If the promise is rejected, the test will fail.

For example, let's say that fetchData returns a promise that is supposed to resolve to the string 'peanut butter'. We could test it with:

```
test('the data is peanut butter', () => {
  return fetchData().then(data => {
    expect(data).toBe('peanut butter');
  });
});
```

- Async/Await:

Alternatively, you can use async and await in your tests. To write an async test, use the async keyword in front of the function passed to test. For example, the same fetchData scenario can be tested with:

```
test('the data is peanut butter', async () => {
  const data = await fetchData();
  expect(data).toBe('peanut butter');
});

test('the fetch fails with an error', async () => {
  expect.assertions(1);
  try {
    await fetchData();
  } catch (error) {
    expect(error).toMatch('error');
  }
});
```

You can combine async and await with .resolves or .rejects.

```
test('the data is peanut butter', async () => {
  await expect(fetchData()).resolves.toBe('peanut butter');
});

test('the fetch fails with an error', async () => {
  await expect(fetchData()).rejects.toMatch('error');
});
```

In these cases, async and await are effectively syntactic sugar for the same logic as the promises example uses.

- Callbacks

If you don't use promises, you can use callbacks. For example, let's say that fetchData, instead of returning a promise, expects a callback, i.e. fetches some data and calls callback(null, data) when it is complete. You want to test that this returned data is the string 'peanut butter'.

By default, Jest tests complete once they reach the end of their execution. That means this test will *not* work as intended:

```
// Don't do this!
test('the data is peanut butter', () => {
  function callback(error, data) {
    if (error) {
      throw error;
    }
    expect(data).toBe('peanut butter');
  }

  fetchData(callback);
});
```

The problem is that the test will complete as soon as fetchData completes, before ever calling the callback.

There is an alternate form of test that fixes this. Instead of putting the test in a function with an empty argument, use a single argument called done. Jest will wait until the done callback is called before finishing the test.

```
test('the data is peanut butter', done => {
  function callback(error, data) {
    if (error) {
      done(error);
      return;
    }
    try {
      expect(data).toBe('peanut butter');
      done();
    } catch (error) {
      done(error);
    }
  }

  fetchData(callback);
});
```

If done() is never called, the test will fail (with timeout error), which is what you want to happen.

If the expect statement fails, it throws an error and done() is not called. If we want to see in the test log why it failed, we have to wrap expect in a try block and pass the error in the catch block to done. Otherwise, we end up with an opaque timeout error that doesn't show what value was received by expect(data).