NodeJS Unit Test Best Practices

For every NodeJS project, one need to use either TDD or BDD approach. The following are the tools that are required to perform tests in NodeJS project.

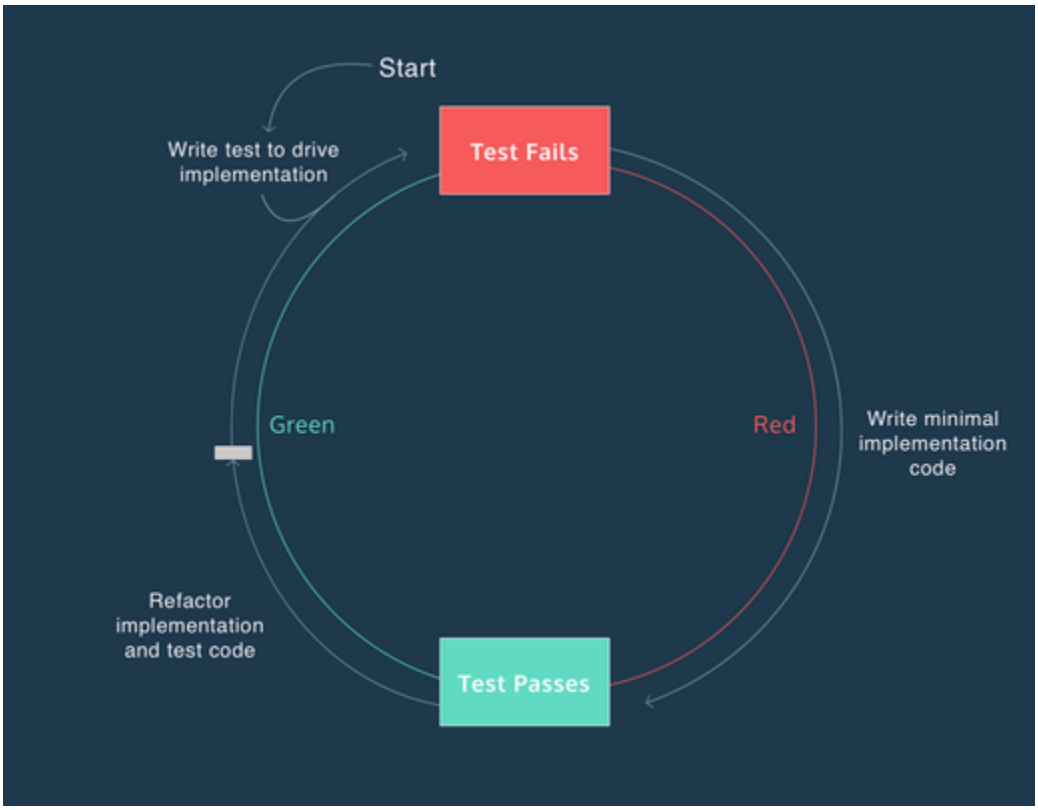
1. Jest
2. eslint-plugin-jest

For every test, we will use Jest as the test framework and code coverage. And such tools already provided by generator-astro by default.

There are several steps for testing pipeline. But, what we want to cover here is the test for coding pipeline which consists:

- Linting
- Contextual test
- Complexity
- Red-green-refactor

Take a look at the following diagram when we are creating a test using TDD.



Developer should always start in red phase, where the first code to write is the test codes. This phase will make the test fail. Then one need to add minimal implementation code and fix the test to green (passes), then refactor again for more tests repeating the cycle until the feature completed.

Every developer need to keep in mind three rules when creating a test:

1. What is being tested, for example a developer is testing BookingService.listBooking method.
2. What is the scenario, for example nothing passed to the method.
3. What is the expected result, for example, the booking list will be returned.

Sample code

```
// 1. UUT (Unit Under Test)
describe('Booking Service', () => {
  describe('listBooking', () => {

 // 2. Scenario
 it('should return booking list', () => {
  const response = new BookingService().list();

  // 3. Expectation
  expect(response.bookings).toEqual(expect.any(Array));
    });

  });
});
```

The above test, we can easily understand the context of the test is for testing `BookingService.listBooking`. The test case also clearly mention about testing the return of the booking list.

Use BDD-style assertions

A developer must use declarative BDD style using ***expect*** or ***should*** and avoid using any custom code. The reason we are to use BDD-style assertion is to be able to understand what the test is about without deep reading the codes. Make every test case as short as possible.

Lint with eslint plugins for test framework

In our case we will use eslint-plugin-jest. The reason for the use of test linter, is to make a good habit to confirm with best practices on test development, like avoiding `skip()` that will lead to false reports.

This plugin is available by default in generator-astro.

Black-box testing

One should always test the public methods, there's no need to test private methods, except when one really required to do so. Because once a developer test the public methods, then the private methods will be tested. So, make sure that the public method call the private method. No room for unused private implementations.

It is considered anti-pattern if one create a test for a private method being called by public method in the same file or class. Or by testing a method where the end-user never call, where this can be public method but not in the specification for the end-user consumption, and this is called over-specifications in which a developer must avoid.

Choose test doubles wisely

There are various ways to make a test double. One can use mock, stubs or spies. But, a developer must be conscious about the overhead of such test double if used in their test cases. One should not use mock if stub can be used instead. This also applies to stub vs spies, where a developer should avoid stub if they only need to use spy on some method calls. Test doubles are very handy and powerfull sometimes, but one can abuse it to the extend will make the test cases overhead. This overhead might not noticeable in small project, but in bigger project and multi-team environment this will cause a problem where there are hundreds of thousands of test cases being run on the pipeline. So, don't make a test case a burden, but make it a helpful friend.

One should avoid mock as much as one can, unless one deemed it a necessary to do so. Most of the time, stub and spies are enough for a developer to satisfy the test cases.

You can have a look of a bad example on spying on certain method using mock.

Anti-pattern

```
const userService = require('./user-service.js');
jest.mock('./user-service.js');

...
const userId = 1;

test('when a user deleted, make sure cleanup called once', async () => {
 userService.delete(userId)
  .then(() => {
   expect(userService.cleanup).toBeCalledTimes(1);
  });
});

...
```

Another example of best way to spy on method, without mocking the whole object.

Good-pattern

```
...
const userId = 1;

test('when a user deleted, make sure cleanup called once', async () => {
 const cleanupSpy = jest.spyOn(UserService.prototype, 'cleanup');
 new UserService().delete(userId)
  .then(() => {
   expect(cleanupSpy).toBeCalledTimes(1);
  });
});

...
```

Both implementations looks ok, but the latter is much better, because it does not have overhead, and one can focus only on the method one interested in, without mocking the whole object.

Don't use "common" data

If one need to provide same fake data to the test, one should always avoid using data that is only valid to the method being tested. Because using such data will make test cases green in development machine, but might fail in production environment. One should provide a abusive strange parameters to simulate if the end-user put in some crazy parameter, so one can see if the method can handle such strange parameter or not.

For example when one uses "**foo**" as parameter in a test case, might make the test case pass as green. But this might fail if one pass the abusive string like a very long string, the kind of exploit might use when fuzzing the service.

Bad example

```
...
test('when describe a user, should return valid data', async () => {
 const result = userService.describe(1);
 expect(result).toHaveProperty('id');
 ...
});
...
```

The above test case will be green on development machine, because the developer provide it with valid number.

Good example

```
...
test('when describe a user, should return valid data', async () => {
 const result = userService.describe(faker.random.number()); // random
number: 9999999999999
 expect(result).toHaveProperty('id');
 ...
});
...
```

The above test will make sure the method being tested is checking for boundary for the parameter, otherwise the test case will be red.

So, using abusive parameter is very important to prevent a hacker from abusing the parameter of the service/app on the production environment. Better be ready than to regret.

Fuzz testing your input

For some critical feature that serves public purposes, a developer need to make sure that the feature can return valid responses for every different combination of inputs.

For this purpose one can use generative test, for jest a developer can use jasmine-check.