DAOUTECH

Vue Test Utils(VTU)과 Jest를 이용한

Vue.js 단위 테스트

플랫폼서비스팀

김설한

2021.06.28

• 목차

01

라이브러리 소개

라이브러리 소개

1)Vue Test Utils (VTU)

- Vue.js 공식 단위 테스트 유틸리티 라이브러리 (단위 테스트 단순화 유틸리티 함수 집합)
- 공식 가이드를 제공 하고 있다. (https://vue-test-utils.vuejs.org/)

Vue Test Utils: Introduction

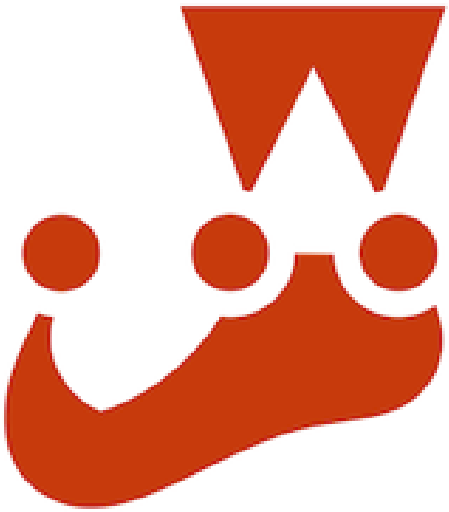
Utilities for testing Vue components. ... Introduction. **Vue Test Utils** is the official unit testing utility library for Vue.js. This is the documentation for **Vue Test Utils** v1, ...

Wrapper · Mount() · API · Guides

- Vue 컴포넌트의 구성 요소를 탑재하고, 상호 작용 할 수 있게 해 주는 메서드를 제공해준다.
  1) 컴포넌트 렌더링 함수
  2) 컴포넌트 요소 접근 함수
  3) 트리거 함수
  4) 데이터 세팅 함수

라이브러리 소개

2) Jest



- 페이스북에서 만든 **JS 테스팅 라이브러리**.
- 다른 JS 테스팅 라이브러리와 다르게,

  Test Runner와 Test Mathcer, Test Mock 프레임워크

  까지 제공해주어 All-in-one으로 사용 가능.
- Vue.js 환경 단위 테스트 공식 라이브러리

  Vue Test Utils(VTU)에서 권장하는 테스트 러너.

02

설치 및 환경 설정

설치 및 환경 설정

1) 라이브러리 설치

```
npm install jest @vue/test-utils vue-jest babel-jest --save-dev
```

2) package.json에 설정 내용 추가

```
"scripts": {
  "serve": "vue-cli-service serve --port 9000",
  "build": "vue-cli-service build",
  "test:unit": "vue-cli-service test:unit",
  "lint": "vue-cli-service lint",
  "test": "jest"
},
```

- scripts에 test로 jest로 등록
  (npm test 명령어로 테스트 실행 가능)

```
"jest": {
  "moduleFileExtensions": [
    "js",
    "ts",
    "json",
    "vue"
  ],
  "transform": {
    ".*\\.(vue)$" : "vue-jest"
  }
}
```

- moduleFileExtensions : Jest가 처리할 파일의 확장자
- transform : .vue 파일을 "vue-jest"로 처리하겠다는 의미

설치 및 환경 설정

3) .eslintrc.js에 jest 설정 추가

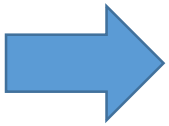
```
env: {
    browser: true,
    node: true,
    jest: true,
},
```

- 구문과 스타일을 체크하는 esLint에 jest를 사용을 알림

4) 설정 테스트

- test.js라는 파일을 생성하고, 테스트 코드를 작성한 후 npm test로 설정 확인.

```
test("1 is 1", () => {
  expect(1).toBe(1);
});
```



```
$ npm test

> my-jest@1.0.0 test /my-jest
> jest

 PASS  ./test.js
  ✓ 1 is 1 (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.868s, estimated 1s
Ran all test suites.
```

- Jest는 **파일명이 test.js로 끝**나거나, **__test__ 디렉터리 하위에 있는** 파일을 테스트 코드로 인식하고 실행 시킨다.
- 특정 테스트 파일만 실행하고 싶은 경우에는 **npm test <파일명 or 경로>** 를 사용하면 된다.

03

Vue Test Utils(VTU) 함수

Vue Test Utils(VTU) 함수

1) 컴포넌트 렌더링 함수

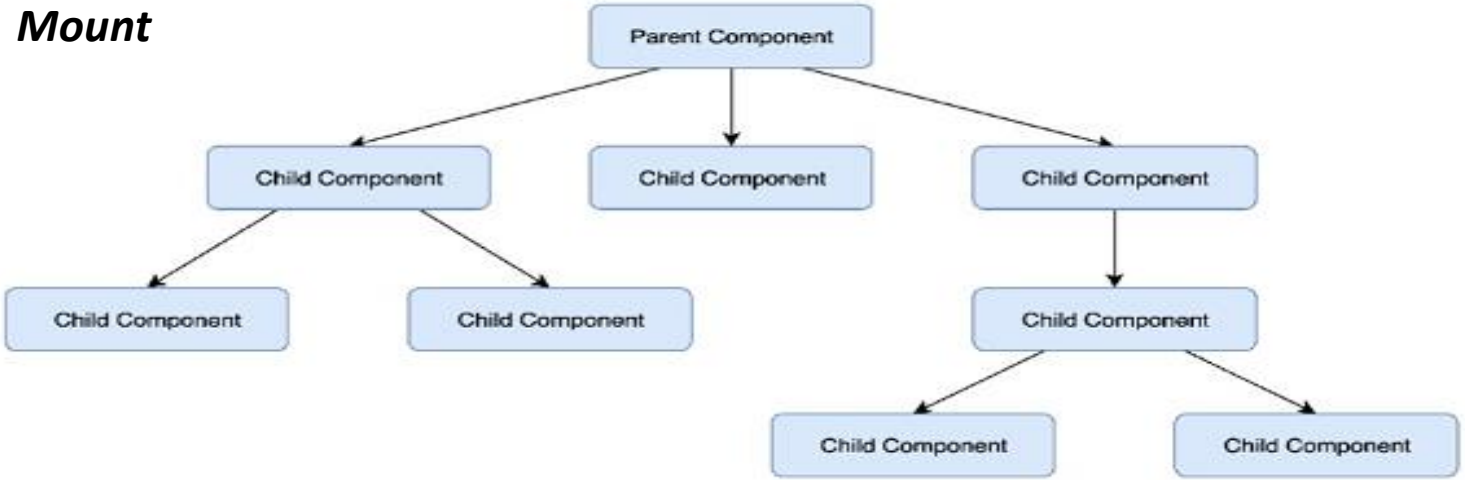
① mount()

- mount한 컴포넌트와 그 안에 포함된 하위 구성요소들을 모두 렌더링 해 return해주는 함수.

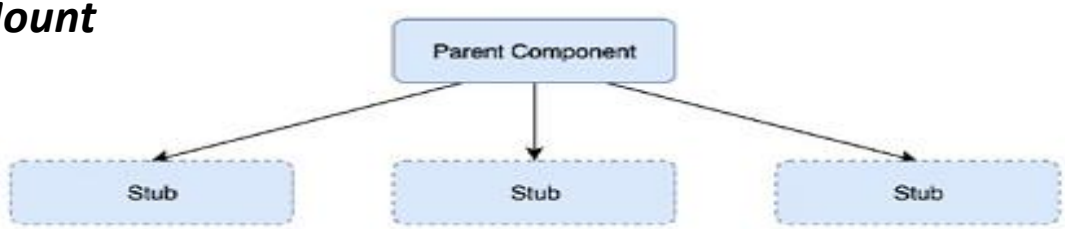
② shallowMount()

- 해당 컴포넌트의 하위 component 들은 렌더링 하지 않고 return해주는 함수.

Mount



shallowMount



Vue Test Utils(VTU) 함수

2) Wrapper를 통한 컴포넌트 요소 접근 함수

① find()

- 컴포넌트 요소를 찾아 return해준다.
- return 받은 컴포넌트 요소에 .을 사용해 속성값에 접근할 수도 있다.
- 인자로 넣은 선택 자에 해당하는 요소를 리턴 해준다.

```
const wrapper = mount(Foo)
```

```
const div = wrapper.find('div')
expect(div.exists()).toBe(true)
```

```
const byId = wrapper.find('#bar')
expect(byId.element.id).toBe('bar')
```

- ref/ name값을 통한 접근은 속성 명을 인자에 기입해 사용한다.

```
const button = wrapper.find({ ref: 'testButton' })
```

Vue Test Utils(VTU) 함수

2) Wrapper를 통한 컴포넌트 요소 접근 함수

② findComponent()

- 컴포넌트의 하위 컴포넌트를 찾아 return해준다.

```
const wrapper = mount(Foo)

const bar = wrapper.findComponent(Bar) // => finds Bar by component instance
expect(bar.exists()).toBe(true)
const barByName = wrapper.findComponent({ name: 'bar' }) // => finds Bar by `name`
expect(barByName.exists()).toBe(true)
const barRef = wrapper.findComponent({ ref: 'bar' }) // => finds Bar by `ref`
expect(barRef.exists()).toBe(true)
```

Vue Test Utils (VTU) 함수

3) 트리거 함수 - trigger

- Wrapper DOM 노드에 이벤트를 발생시켜준다.
- Wrapper 자체에 사용할 수도 있고

```
const wrapper = mount(Foo, {
  propsData: { clickHandler }
})
```

```
wrapper.trigger('click')
```

```
await wrapper.trigger('click', {
  button: 0
})
```

이벤트 인자 전달도 가능

- find를 통해 접근한 Wrapper 내부 요소에도 사용 가능하다.

```
const input = wrapper.find('input')
input.element.value = 100
input.trigger('click')
```

Vue Test Utils (VTU) 함수

4) 데이터 setting 함수 - setData, setProps

➢ Wrapper의 Data와 PropsData를 세팅해준다.

```
wrapper.setData({ foo: 'bar' })
wrapper.setProps({ foo: 'bar' })
```

- wrapper(컴포넌트)의 data인 foo를 'bar'로 setting.
- wrapper(컴포넌트)의 propsData인 foo를 'bar'로 setting.

➢ 또한, Data와 PropsData는 Wrapper 초기화 시에도 설정할 수 있다.

```
const wrapper = mount(Component, {
  data() {
    return {
      bar: 'my-override'
    }
  }
})
```

```
const wrapper = mount(Component, {
  propsData: {
    msg: 'aBC'
  }
})
```

04

Jest Matcher 함수

Jest Matcher 함수

1) toBe()

- 숫자나 문자/Boolean 같은 기본형(primitive)값을 비교.

2) toEqual()

- 객체 값을 비교.

```
test("return a user object", () => {
  expect(getUser(1)).toEqual({
    id: 1,
    email: `user1@test.com`,
  });
});
```

3) toHaveLength()

- 배열 길이 체크

```
test("array", () => {
  const colors = ["Red", "Yellow", "Blue"];
  expect(colors).toHaveLength(3);
  expect(colors).toContain("Yellow");
  expect(colors).not.toContain("Green");
});
```

4) toContain()

- 특정 원소가 배열에 들어있는지 확인.

Matcher 함수 앞에 not을 붙이면, 불만족 체크.

05

비동기 코드 테스트

비동기 코드 테스트

```
function fetchUser(id, cb) {
  setTimeout(() => {
    console.log("wait 0.1 sec.");
    const user = {
      id: id,
      name: "User" + id,
      email: id + "@test.com",
    };
    cb(user);
  }, 100);
}
```

0.1초 지연 후,

콜 백 함수(cb)를 호출하는 fetchUser 함수

※ 기본적인 test 방법대로 작성하면, 검증에 실패함

```
test("fetch a user", () => {
  fetchUser(2, (user) => {
    expect(user).toEqual({
      id: 1,
      name: "User1",
      email: "1@test.com",
    });
  });
});
```

FAIL이어야 하는데
PASS 함... ㅠㅠ

```
> npm test

> my-jest@1.0.0 test /my-jest
> jest

 PASS  ./async.test.js
  ✓ fetch a user (1ms)
```

틀린 검증 값 !!!

```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.88s, estimated 1s
Ran all test suites.
```

비동기 코드 테스트

∵ Jest Runner는 테스트 함수를 최대한 빨리 호출하려고 하기에,

콜백 함수는 물론, 그 내부 toEqual() Matcher함수도 기다리지 않는다.

∴ 비동기 코드를 테스트 할 때엔, 콜백 함수 추가 검증이나, 함수 종료 대기를 명시해야 한다.

① 콜백 함수로 구현된 비동기 코드 테스트

- test 함수의 호출 함수 제일 마지막 부분에, done() 함수를 명시해,
  콜백 함수 호출과 추가 검증을 알린다.

```
test("fetch a user", (done) => {
  fetchUser(2, (user) => {
    expect(user).toEqual({
      id: 1,
      name: "User1",
      email: "1@test.com",
    });
    done();
  });
});
```

```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.894s, estimated 1s
Ran all test suites.
```

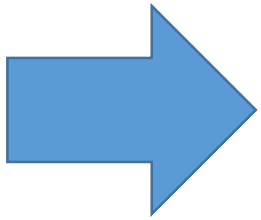
비동기 코드 테스트

② Promise로 구현된 비동기 코드 테스트

```
function fetchUser(id) {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("wait 0.1 sec.");
      const user = {
        id: id,
        name: "User" + id,
        email: id + "@test.com",
      };
      resolve(user);
    }, 100);
  });
}
```

- 0.1초 지연 후, Promise 객체를 리턴하는 함수
- Promise로 구현된 비동기 코드에 return문을 추가하면, 리턴된 Promise가 resolve될 때 까지 Jest Runner가 기다린다.

```
test("fetch a user", () => {
  return fetchUser(2).then((user) => {
    expect(user).toEqual({
      id: 1,
      name: "User1",
      email: "1@test.com",
    });
  });
});
```



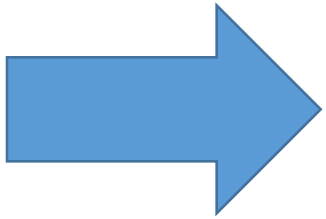
```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        1.021s
Ran all test suites.
```

비동기 코드 테스트

③ async/await으로 구현된 비동기 코드 테스트

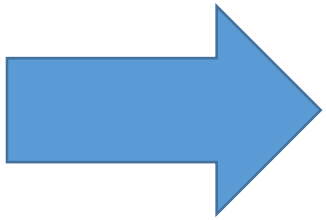
➢ async/await으로 구현된 비동기 코드는 테스트 함수 맨 앞에 async를 추가하고, Promise를 리턴 하는 함수 앞에 await을 붙여주면 된다. (기본 async/await 사용법과 동일)

```
test("fetch a user", async () => {
  const user = await fetchUser(2);
  expect(user).toEqual({
    id: 2,
    name: "User2",
    email: "2@test.com",
  });
});
```



```
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.476s
Ran all test suites.
```

```
test("fetch a user", async () => {
  const user = await fetchUser(2);
  expect(user).toEqual({
    id: 1,
    name: "User1",
    email: "1@test.com",
  });
});
```



```
Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        0.997s, estimated 1s
Ran all test suites.
```

06

테스트 전/후 처리

Jest 테스트 전/후 처리

➢ Jest는 테스트 코드 작성에 공통으로 필요한 작업들을 작성하는 메소드를 제공한다.

① beforeEach() : 각 테스트 함수가 실행되기 전에 호출되는 코드

```
beforeEach(() => {
  data.users.push(
    { id: 1, email: "user1@test.com" },
    { id: 2, email: "user2@test.com" },
    { id: 3, email: "user3@test.com" }
  );
});
```

- 매 테스트가 수행되기 전에 호출되므로, 테스트 데이터를 입력하는 등 전 처리 코드를 작성해 두면 유용하다.

② afterEach() : 각 테스트 함수를 실행한 이후에 호출되는 코드

```
afterEach(() => {
  data.users.splice(0);
});
```

- 매 테스트가 수행된 후 호출되므로, 테스트 데이터를 정리하는 등 마무리 코드를 작성해 두면 유용하다.

Jest 테스트 전/후 처리

③ beforeAll(), afterAll()

- beforeEach(), afterEach()와 달리, 전체 테스트 함수 실행의 이전과 이후에 한번 씩 호출
- 대표적인 예 : DB connection 연결/종료

```
let connection;

beforeAll(() => {
  conection = openConnection({ host: "...", port: "..." });
});

afterAll(() => {
  conection.close();
});
```

④ only(), skip()

- 테스트 코드를 디버깅 할때 유용한 함수들.
- only() : 테스트 파일 內 하나의 함수만 단독 실행.
- skip() : 테스트 파일 內 해당 함수만 제외하고 실행.

```
test.only("run only", () => {
  // 이 테스트 함수만 실행됨
});

test("not run", () => {
  // 실행 안됨
});
```

```
test.skip("skip", () => {
  // 이 테스트 함수는 제외됨
});

test("run", () => {
  // 실행됨
});
```

Jest 테스트 전/후 처리

⑤ describe() : 테스트 파일에 테스트 함수가 多수 작성되어 있는 경우,
관련된 테스트 함수들 끼리 그룹화.

```
describe("group 1", () => {
  test("test 1-1", () => {
    // ...
  });

  test("test 1-2", () => {
    // ...
  });
});

describe("group 2", () => {
  it("test 2-1", () => {
    // ...
  });

  it("test 2-2", () => {
    // ...
  });
});
```

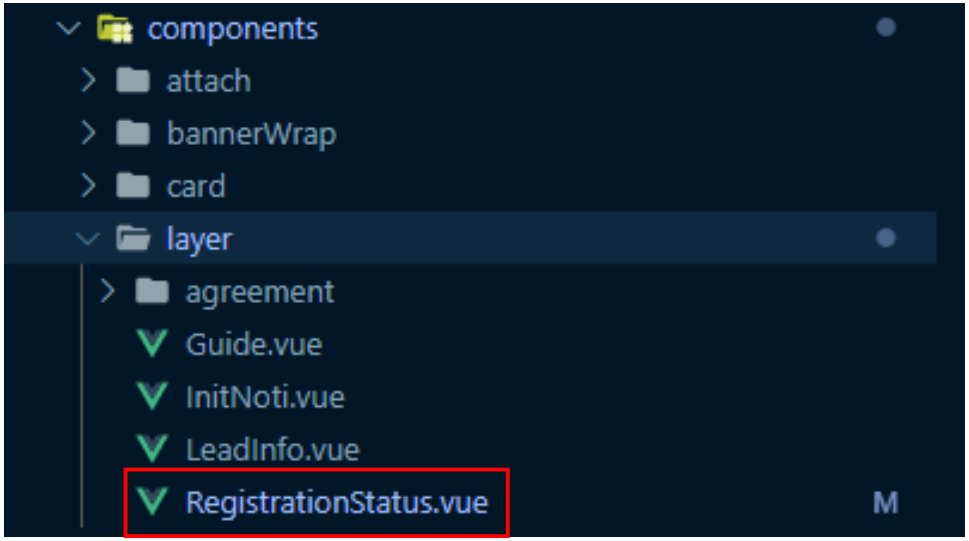
⑥ it() : test()와 완전히 동일한 기능을 하는 함수.
기존에 많이 사용되었던 테스트 라이브러리인 Mocha, Jasmin에서
함수명을 it()으로 사용하였기에 Jest에서도 it()을 test()의 별칭으로 제공해 주고 있음.

08

VTU와 Jest로 테스트 코드 작성 해 보기

테스트 컴포넌트 소개

1) 테스트 컴포넌트



example.spec.js U | RegistrationStatus.vue M | constants.js

src > utils > constants.js > default > partnerContactInfo

```
export default Object.freeze({
    targetTypeDesc: {
        user: '사용자',
        department: '부서',
        position: '직위',
        grade: '직급',
        duty: '직책',
    },
    partnerContactInfo: {
        // 파트너사 연락처 (key : id, value : 데이터)
        9: {
            contact: '050-7600-4009',
            email: 'sales@donutbook.com',
            watch: '오전 10시 ~ 오후 5시 (월~금)',
        },
    },
});
```

2) 컴포넌트의 data와 propsData

```
props: {
    service: {
        type: Object,
    },
},
data() {
    return {
        partnerContactInfo: this.$constants.partnerContactInfo[this.service.partnerId],
    };
},
```

테스트 컴포넌트 소개

3) 테스트 내용

① ID가 name, contact인 요소의 text 검증

```
<div class="subject">
    <strong id="name">{{ service.name }}</strong>
    <span class="txt"> 링크<br />승인 대기 중입니다.</span>
</div>
```

```
<li>
    <span class="ic24 ic_phone"></span>
    <span class="txt" id="contact"
        >연락처 : {{ partnerContactInfo.contact }}</span
    >
</li>
```

테스트 컴포넌트 소개

3) 테스트 내용

② isStepOver 함수 검증

```
isStepOver(step) {
    return statusStep[step] <= statusStep[this.service.registrationStatus];
},
```

```
const statusStep = {
    REGISTRATION_APPLY: 2,
    REGISTRATION_COMPLETE: 3,
};
```

테스트 컴포넌트 소개

3) 테스트 내용

③ async/await으로 구현된 reApply 함수 검증

```
async reApply() {
    let result;
    const params = {
        linkProductId: this.service.id,
        additionalService: this.service.name,
        termsProvided: true,
    };
    await applyService(params)
        .then(res => {
            this.$toasted.show('신청이 완료되었습니다.');
            this.closeTerm();
            result = true;
        })
        .catch(error => {
            this.$toasted.show(error.data.message);
            result = false;
        });

    return result;
},
```

테스트 수행

- test-utils와 RegistrationStatus 컴포넌트를 import 해준다.

```
import { shallowMount } from '@vue/test-utils';
import RegistrationStatus from '@/components/layer/RegistrationStatus.vue';
```

- 단위 테스트들에서 공통으로 사용할 wrapper를 전역 변수로 선언하고,
  전체 테스트 실행 전 wrapper를 초기화 해준다. (propsData도 함께 초기화)

```
let wrapper;
beforeAll(() => {
    wrapper = shallowMount(RegistrationStatus, {
        propsData: {
            service: {
                id: 9,
                name: '알파몰',
                partnerName: '알파(주)',
                partnerId: 9,
                registrationStatus: 'REGISTRATION_COMPLETE',
            },
        },
    });
});
```

테스트 수행

```
describe('RegistrationStatus 테스트', () => {
    it('text 확인', () => {
        expect(wrapper.find('#name').text()).toMatch('알파몰');
        // id가 name인 요소를 가져와 text 검증
        expect(wrapper.find('#contact').text()).toMatch('050-7600-4009');
        // id가 contact인 요소를 가져와 text 검증
    });

    it('isStepOver 함수 테스트', () => {
        expect(wrapper.vm.isStepOver('REGISTRATION_APPLY')).toBe(true);
        // vm을 통해 함수 접근 - return  값 true 검증
    });

    it('비동기 함수 reApply 테스트', async () => {
        // async-await 비동기 함수는 테스트 문에도 async-await 사용
        const response = await wrapper.vm.reApply();
        expect(response).not.toBe(true); // return 값이 true가 <아님(not)>을 검증
    });
});
```

```
PASS tests/unit/test.js
  RegistrationStatus 테스트
    √ text 확인 (2 ms)
    √ isStepOver 함수 테스트
    √ 비동기 함수 reApply 테스트 (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        4.162 s
Ran all test suites.
```

마무리

[장점]
1. VTU, Jest 제공 함수들을 통해, 테스트를 자동화 할 수 있다.
2. 테스트 파일만 따로 작성해 둔다면 environment처럼 사용할 수 있다.

[단점]
1. Vue Test Utils와 Jest에 대한 학습이 필요하다.
2. 언급하지 않았지만, babel 버전이나 Vuetify / ESLint 사용 유무 등에 따라 에러가 발생할 수 있다.
   따라서, 환경 구축에 오랜 시간이 소요되는 단점이 있다.

충분한 학습 시간을 가지고 개발 일정이 촉박하지 않을 때.....
개발 환경 구축 시 테스트 환경도 함께 구축하여 사용하는 게 좋을 것 같다.

(참고)
- Vue Test Utils 공식 가이드 : https://vue-test-utils.vuejs.org/
- HEROPY Tech 블로그 : https://heropy.blog/2020/05/20/vue-test-with-jest/

참고

Mocking

Jest Mocking

1) Mocking이란?

- 단위 테스트를 작성할 때, 코드가 의존하는 부분을 가짜(mock)으로 대체하는 기법.
- 특정 기능만 분리해서 테스트 하고자 하는, 단위 테스트의 근본적인 사상에 맞춘 기법.
- Mocking을 이용하면, 실제 객체를 사용하는 것 보다 가볍고 빠르게 실행하며, 항상 동일한 결과를 내는 테스트를 작성할 수 있다.

Jest가 제공하는 Mocking 관련 함수

1) jest.fn()

```
const mockFn = jest.fn();
```

```
mockFn();
mockFn(1);
mockFn("a");
mockFn([1, 2], { a: "b" });
```

위 가짜 함수의 호출 결과는 모두 undefined인데, 어떤 값을 리턴해야 할 지 설정해 주는 함수가

2) mockReturnValue(리턴값) 이다.

```
mockFn.mockReturnValue("I am a mock!");
console.log(mockFn()); // I am a mock!
```

3) mockResolvedValue(리턴값)

```
mockFn.mockResolvedValue("I will be a mock!");
mockFn().then((result) => {
  console.log(result); // I will be a mock!
});
```

가짜 비동기 함수의 Promise resolve 값을 지정.

Jest가 제공하는 Mocking 관련 함수

4) mockImplementation(코드 즉석 구현)

```
mockFn.mockImplementation((name) => `I am ${name}!`);
console.log(mockFn("Dale")); // I am Dale!
```

- 실제 구현 함수를 제거하고, 함수의 구현을 즉석으로 바꿀 수 있다.

```
mockFn("a");
mockFn(["b", "c"]);

expect(mockFn).toBeCalledTimes(2);
expect(mockFn).toBeCalledWith("a");
expect(mockFn).toBeCalledWith(["b", "c"]);
```

- 테스트 작성 시, 가짜 함수가 유용한 점은 자신이 어떻게 호출 되었는지(호출 횟수, 파라메타) 를 모두 기억한다는 점이다.
- 위 예시처럼, 가짜 함수 용으로 설계된 Jest Matcher **toBeCalled*** ** 함수를 사용하면 가짜 함수가 몇 번 호출되었고, 인자로 어떤 값이 넘어왔었는지 검증할 수 있다.

Jest가 제공하는 Mocking 관련 함수

5) jest.spyOn(object, methodName)

해당 함수의 호출 여부와 어떻게 호출 되 었는 지만 알아내는 함수.

```
const calculator = {
  add: (a, b) => a + b,
};

const spyFn = jest.spyOn(calculator, "add");

const result = calculator.add(2, 3);

expect(spyFn).toBeCalledTimes(1);
expect(spyFn).toBeCalledWith(2, 3);
expect(result).toBe(5);
```

- 함수의 구현을 가짜로 대체할 필요는 없고, 호출 관련 정보만 필요할 때 사용
- jest.fn() 처럼, **toBeCalled*** ** 함수를 통해 함수의 호출 횟수와 인자 값을 검증 할 수 있다.

감사합니다 !