

# Jest, testing-library 를 이용한 React 단위 테스트 작성 방법

- 1. 개요
- 2. Webstorm환경에서의 테스트
  - 2.1 Webstorm 설정
- 3. 테스트 코드 작성
  - 3.1 Jest, Testing Library의 다양한 함수, 문법
    - 3.1.1 Jest
      - 3.1.1.1 기초 문법
      - 3.1.1.2 matcher
      - 3.1.1.3 테스트 전후 설정
    - 3.1.2 Testing-Library
      - 3.1.2.1 쿼리
      - 3.1.2.2 쿼리 예시
      - 3.1.2.3 어떤 쿼리를 써야할까
  - 3.2 Jest, Testing Library를 이용한 테스트코드 작성
    - 3.2.1 버튼 클릭시 글자가 나타나는 컴포넌트 테스트
    - 3.2.2 React Lazy, React Router로 이루어진 SPA 컴포넌트 테스트
      - 3.2.2.1 Testing-Library Async Utilities
    - 3.2.3 Mock함수를 이용한 테스트코드 작성
      - 3.2.3.1 mock함수 사용하기
      - 3.2.3.2 mocking module

## 1. 개요

---

WebStorm환경에서 Jest, Testing-Library를 이용해 테스트코드를 작성하여 React Component 를 검증하는 방법

## 2. Webstorm환경에서의 테스트



---

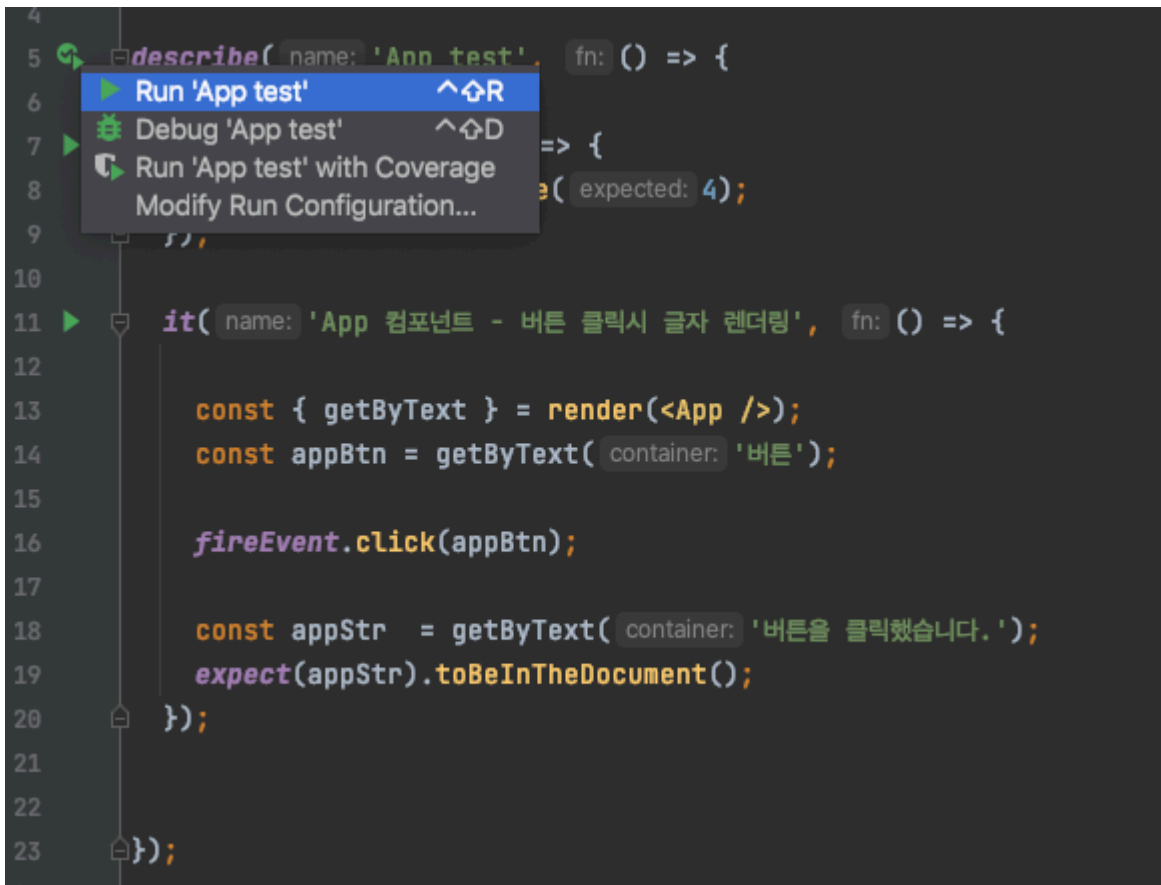
### 2.1 Webstorm 설정

webstorm에서 Jest로 테스트를 실행하고 디버깅할 수 있습니다.

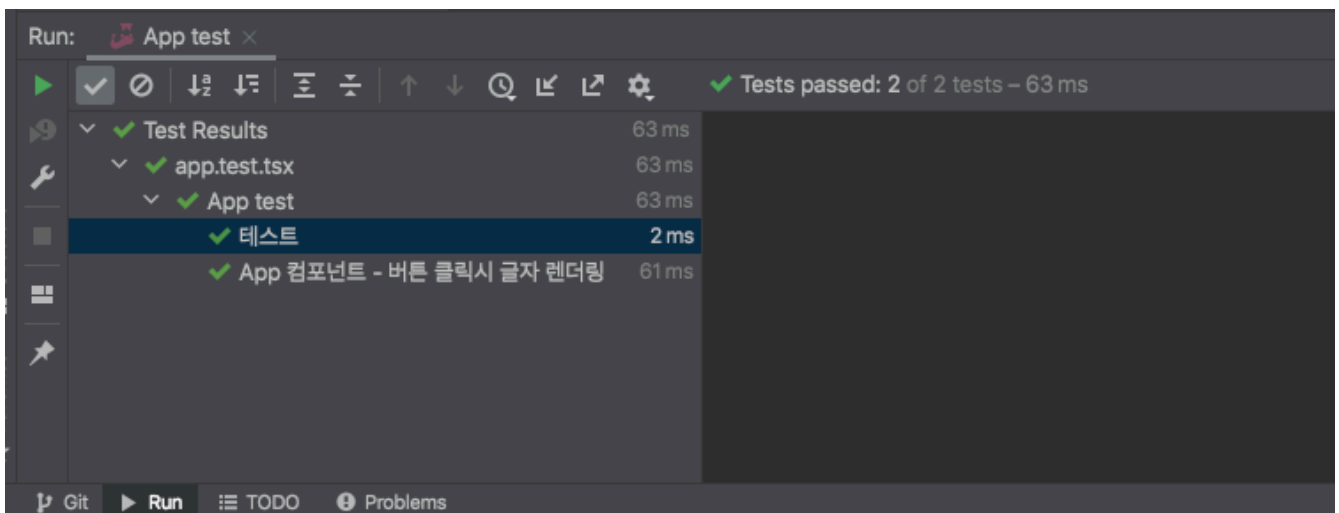
- 테스트 실행

```
app.test.tsx x
1 import React, { Suspense } from 'react';
2 import { getByText, render, screen, waitFor, fireEvent } from '@testing-library/react';
3 import App from './app';
4
5 describe( name: 'App test', fn: () => {
6
7   it( name: '테스트', fn: () => {
8     expect( actual: 2+2 ).toBe( expected: 4 );
9   });
10
11   it( name: 'App 컴포넌트 - 버튼 클릭시 글자 렌더링', fn: () => {
12
13     const { getByText } = render( <App /> );
14     const appBtn = getByText( container: '버튼' );
15
16     fireEvent.click( appBtn );
17
18     const appStr = getByText( container: '버튼을 클릭했습니다.' );
19     expect( appStr ).toBeInTheDocument();
20   });
21
22
23 });
```

webstorm에서 테스트코드를 작성하면 각 테스트마다(it) 옆의 화살표 모양  을 눌러 테스트하거나, 테스트 단위를 묶어둔 제일 큰 단위 (describe)에서  속해있는 모든 테스트를 테스트할 수 있습니다.

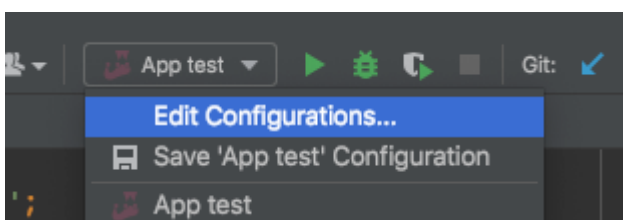


테스트를 진행하면

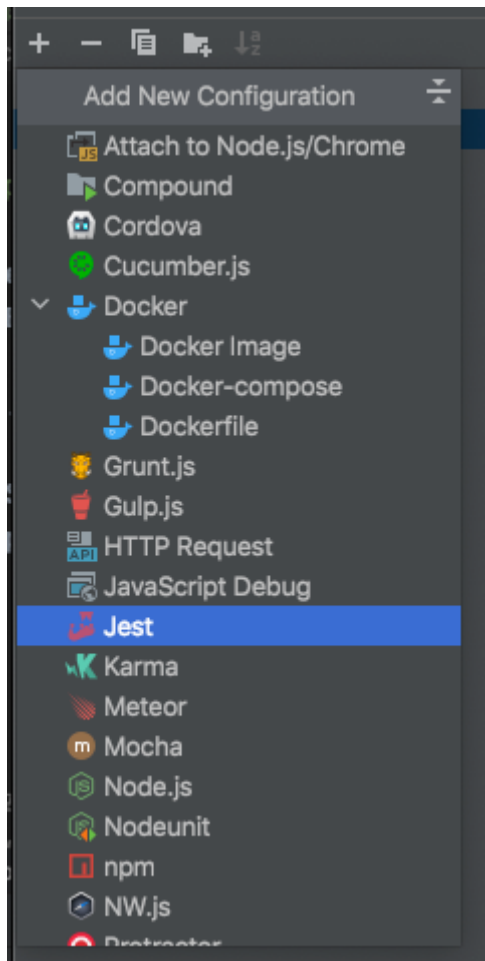
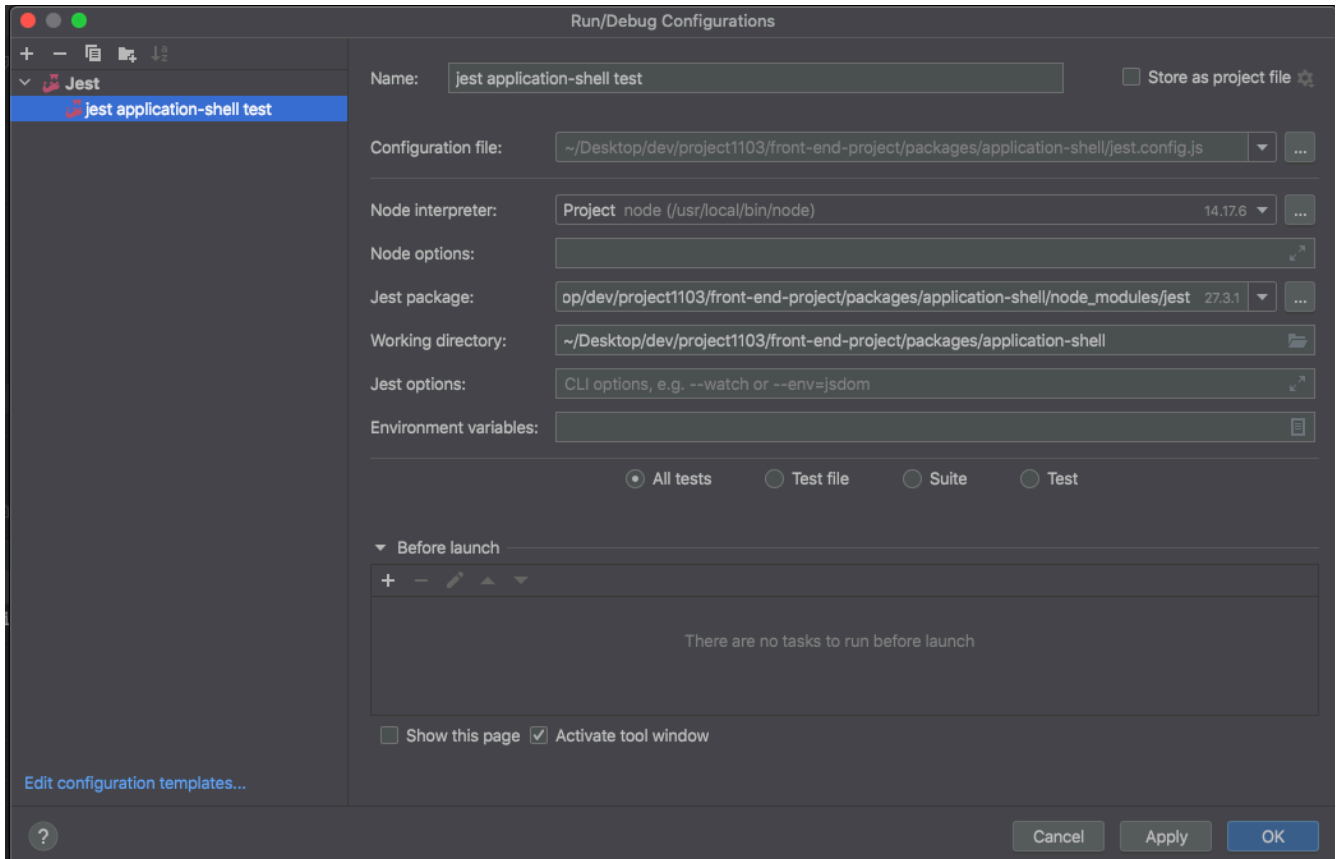


Run에서 테스트한 파일의 이름, 테스트의 이름과 테스트가 통과하였는지 여부가 보여집니다.

- Jest run Configuration 생성



webstorm 우측 상단의 실행환경설정을 클릭합니다.



좌측 상단의 + → Jest를 차례로 선택합니다.

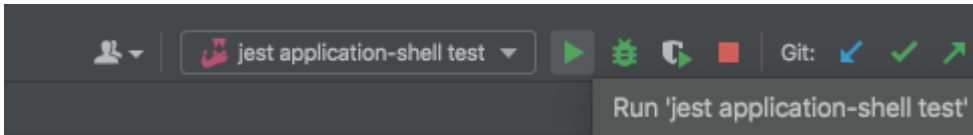
Jest 실행 환경을 설정할 수 있습니다.

**Jest package**에는 다운받았던 Jest 경로를 등록합니다. 프로젝트디렉토리/node\_modules/jest 를 선택하시면 됩니다.

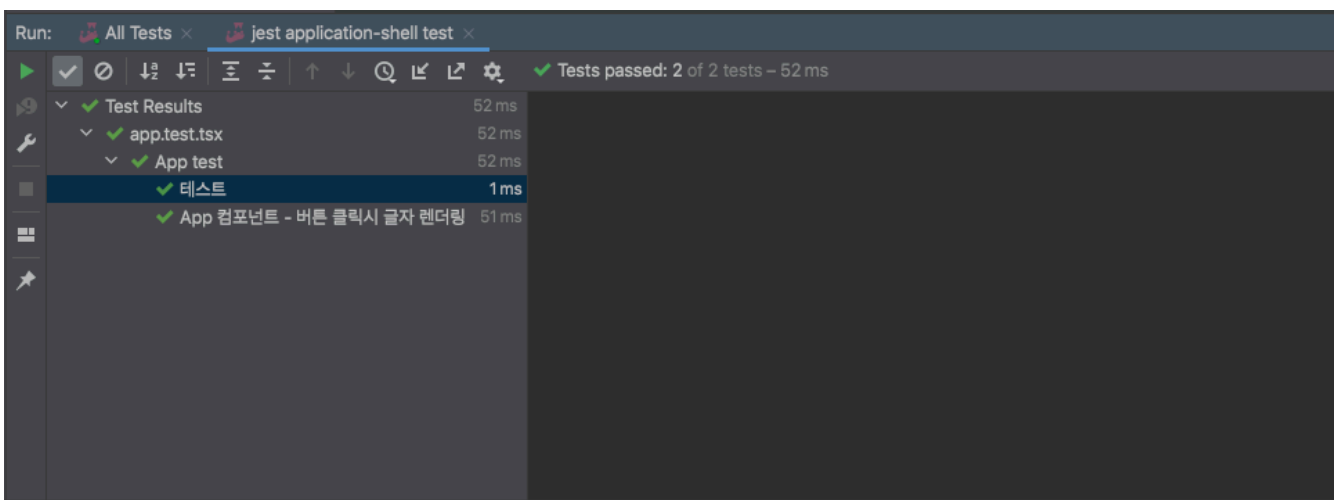
**Working directory**에는 프로젝트 디렉토리를 지정합니다. 보통은 자동으로 지정되어있습니다.

**Jest options**에는 지금 지정하는 WebStorm 실행 환경에서 Jest를 실행할 때 기본적으로 추가될 Jest Options을 등록합니다. --watch 옵션을 넣을 경우 디버깅모드가 끝나지않고 계속 실행상태가 되어 실시간으로 코드 변경을 감지합니다.

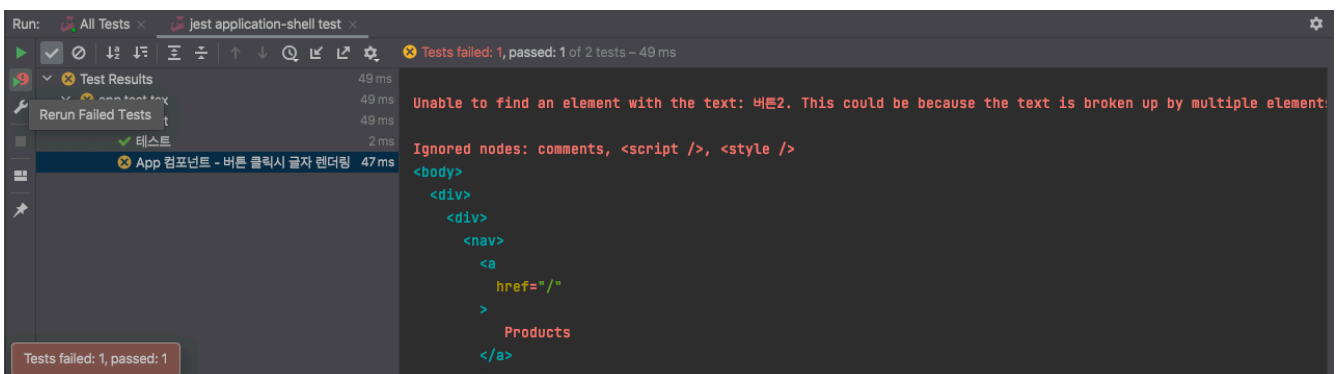
→ 위 설정은 application-shell 폴더에 설치된 Jest package를 사용하고 application-shell의 모든 테스트 파일을 테스트하겠다는 설정입니다.



설정 후 Run 버튼을 누르면



Run에 application-shell 폴더에 있는 모든 테스트 파일이 테스트 됩니다.



테스트가 실패할 경우 실패한 테스트만 실행할 수도 있습니다.

## 3. 테스트 코드 작성

### 3.1 Jest, Testing Library의 다양한 함수, 문법

테스트 코드를 작성하기 전, Jest와 Testing-Library를 사용하기 위한 기초적인 문법입니다.

### 3.1.1 Jest

#### 3.1.1.1 기초 문법

describe: 테스트의 단위를 묶는 가장 큰 단위입니다.

it(): it()을 통해 테스트를 진행합니다. test()도 같은 기능을 제공합니다.

expect(): expect()안에 테스트할 변수나 값을 넣습니다. 이후 toBe나 toEqual을 이용해 예측값과 비교합니다.

```
describe('      ', () => {
  it(' ', () => {
    expect(2+2).toBe(4);
  });
});
```

#### 3.1.1.2 matcher

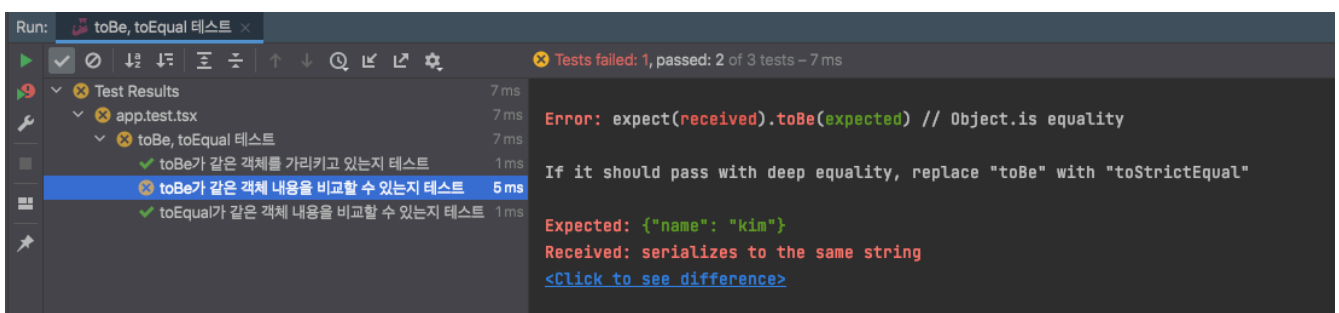
matcher를 이용하면 다양한 방식으로 값을 테스트할 수 있습니다. 많이 쓰이는 matcher에는 toBe, toEqual 등이 있습니다.

```
describe('toBe, toEqual ', () => {
  it('toBe ', () => {
    const obj = {};
    expect(obj).toBe(obj);
  });

  it('toBe ', () => {
    expect({ name: 'kim' }).toBe({ name: 'kim' });
  });

  it('toEqual ', () => {
    expect({ name: 'kim' }).toEqual({ name: 'kim' });
  });
});
```

다음 테스트를 실행하면



'toBe ' .

원시적인타입(number, boolean, string, null)을 사용한다면 toBe와 toEqual에는 차이가 없지만

expect({ name: 'kim' }).toBe({ name: 'kim' }); toBe .

toEqual은 객체 인스턴스 속성을 재귀적으로 확인하기때문에 toEqual은 테스트가 성공합니다.

그외에도 다양한 Matcher 함수가 존재합니다.

```

describe(' ', () => {
  it('null ', () => {
    expect(null).toBeNull() //Null
  });

  it('undefined ', () => {
    expect(undefined).toBeUndefined() //undefined
  });

  it('false ', () => {
    expect(false).toBeFalsy() //false
    expect(0).toBeFalsy()
  });

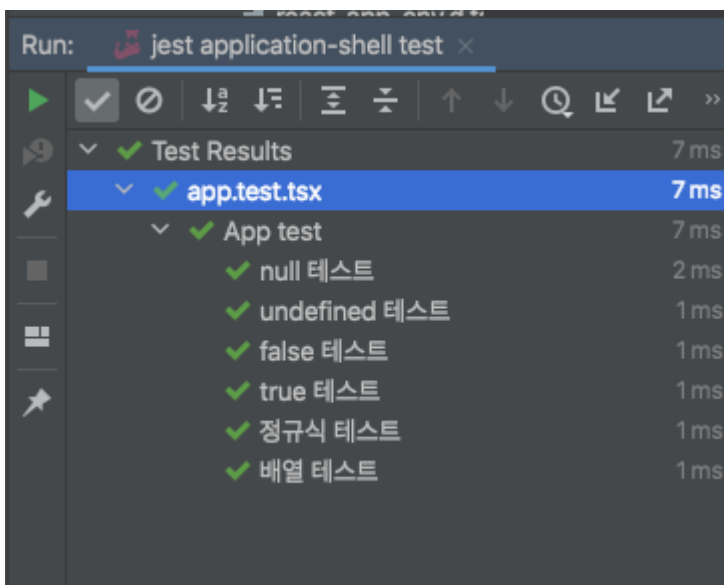
  it('true ', () => {
    expect(true).toBeTruthy() //true
    expect(1).toBeTruthy()
    expect("true").toBeTruthy()
  });

  it(' ', () => {
    expect('000').toMatch(/0*/); // toBe
    toMatch .
  });

  it(' ', () => {
    const arr = [1, 2, 3, 4, 5, '', '']
    expect(arr).toHaveLength(7); //
    expect(arr).toContain(1); //
    expect(arr).toContain('');
  });
});

```

위 테스트코드를 실행하면



모든 테스트가 성공적으로 실행됩니다.

```
it('false ', () => {
    expect(false).toBeFalsy() //false
    expect(0).toBeFalsy()
});

it('true ', () => {
    expect(true).toBeTruthy() //true
    expect(1).toBeTruthy()
    expect("true").toBeTruthy()
});
```

위 테스트가 성공하는 이유는, 느슨한 타입 기반 언어인 자바스크립트는 true와 false가 boolean 타입에 한정되지 않습니다.

따라서 1이 true로 간주되고, 0이 false로 간주되는 것처럼 모든 타입의 값들이 true 아니면 false로 간주하는 규칙이 있습니다.

자바스크립트에서는 false, 0, '', undefined, null, NaN → 6가지 falsy 값이 있습니다. 이를 제외하고 다른 모든 것들은 모두 truthy입니다.

### 3.1.1.3 테스트 전후 설정

테스트를 작성하다보면 모든 테스트 함수에서 공통적으로 필요한 공통 로직이 필요할 때가 있습니다. Jest에서는 테스트 전이나 후에 실행되어야 하는 코드를 작성할 수 있는 기능을 제공합니다.

beforeEach, afterEach -> 각 테스트(it, test) 가 실행되기 전, 후마다 반복적으로 실행됩니다.

beforeAll, afterAll -> 테스트가 실행되기 전, 후마다 한번 실행됩니다.



```
describe('App test', () => {

  beforeEach(() => {
    console.log('  .');
  })

  beforeEach(() => {
    console.log('  .');
  })

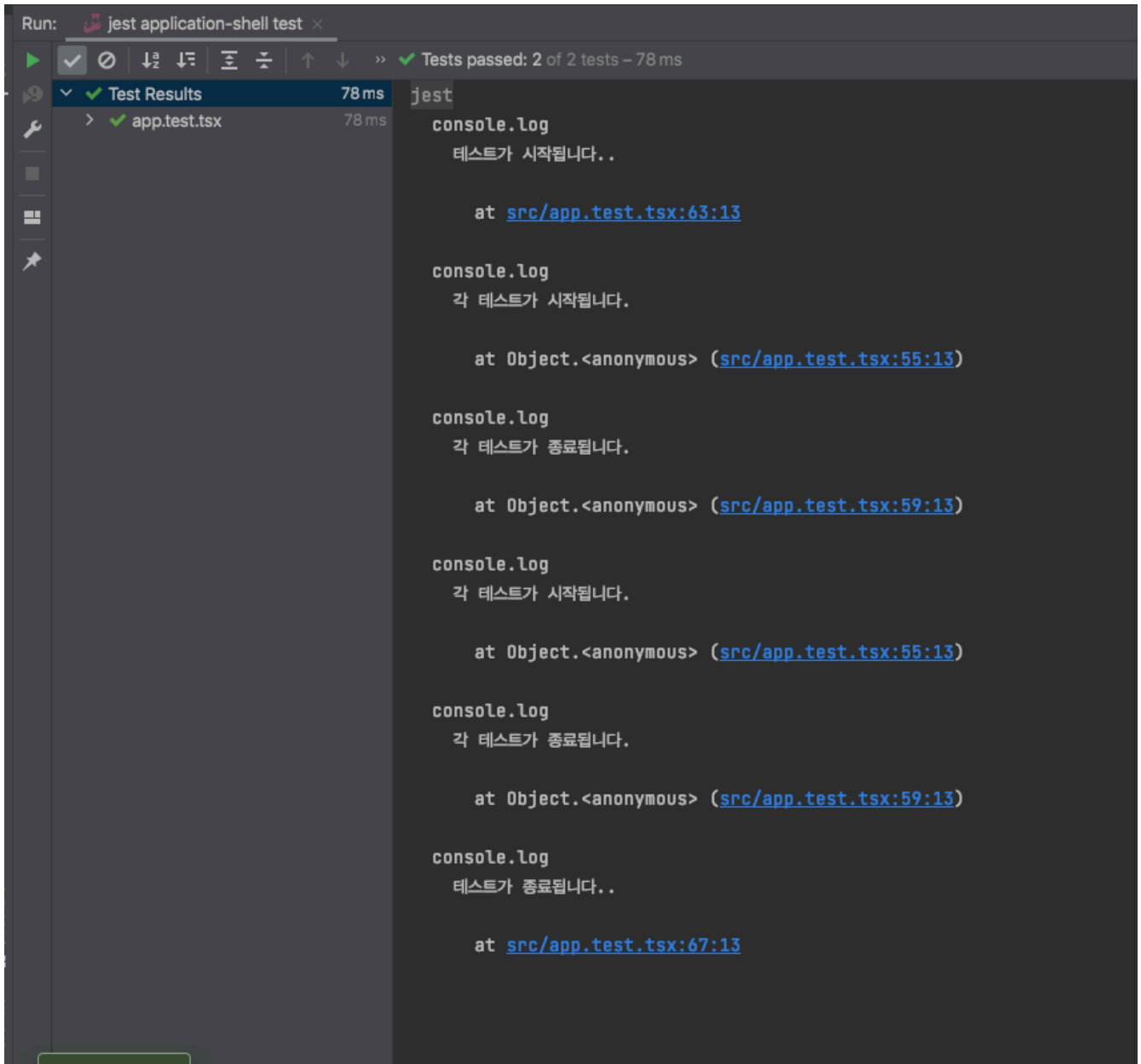
  beforeAll(() => {
    console.log(' ..');
  })

  afterAll(()=>{
    console.log(' ..');
  })

  it('App -      ', () => {
    //
  });

  it('App - <App/> /  ', async () => {
    //
  });

});
```



위 코드를 실행하면 테스트가 한번 실행될 때, 각 테스트가 실행될 때마다 console.log()가 찍히는 것을 확인할 수 있습니다.

테스트 예시입니다.

data.js: 임의의 데이터베이스 역할을 하는 모듈 → 사용자 데이터를 저장하기 위한 users 배열을 가지고 있습니다.

userService.js: data 모듈에 저장되어 있는 users 배열에 사용자 데이터를 조회/생성/삭제/변경하는 기능을 제공하는 모듈입니다.

data.js

```
module.exports = {  
  users: [],  
};
```

userService.js

```

const data = require("./data");

module.exports = {
  findAll() { //
    return data.users;
  },

  create(user) { //
    data.users.push(user);
  },

  delete(id) { //
    data.users.splice(
      data.users.findIndex((user) => user.id === id),
      1
    );
  },

  update(id, user) { //
    data.users[data.users.findIndex((user) => user.id === id)] = user;
  },
};

```

userService 모듈을 테스트해보겠습니다.

```

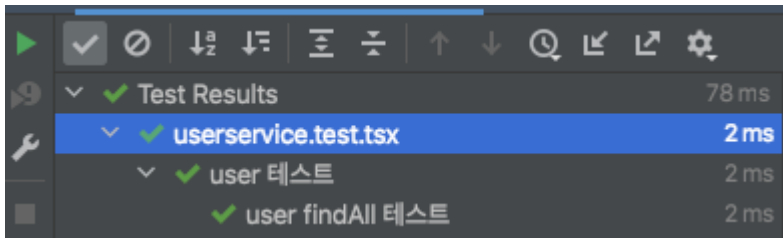
const userService = require("./userService");
const data = require("./data");

describe('user ', () => {
  it("user data findAll ", () => {
    //data  users  3
    data.users.push(
      { id: 1, email: "user1@naver.com" },
      { id: 2, email: "user2@naver.com" },
      { id: 3, email: "user3@naver.com" }
    );

    //  userService.findAll()
    const users = userService.findAll();

    expect(users).toHaveLength(3);
    expect(users).toContainEqual({ id: 1, email: "user1@naver.com" });
    expect(users).toContainEqual({ id: 2, email: "user2@naver.com" });
    expect(users).toContainEqual({ id: 3, email: "user3@naver.com" });
  });
});

```



테스트가 이상없이 통과했습니다.

```
const userService = require("../userService");
const data = require("../data");

describe('user ', () => {
  it("user findAll ", () => {
    //data users 3
    data.users.push(
      { id: 1, email: "user1@naver.com" },
      { id: 2, email: "user2@naver.com" },
      { id: 3, email: "user3@naver.com" }
    );

    // userService.findAll()
    const users = userService.findAll();

    expect(users).toHaveLength(3);
    expect(users).toContainEqual({ id: 1, email: "user1@naver.com" });
    expect(users).toContainEqual({ id: 2, email: "user2@naver.com" });
    expect(users).toContainEqual({ id: 3, email: "user3@naver.com" });
  });

  it("user create ", () => {
    const user = { id: "4", email: "user4@naver.com" };

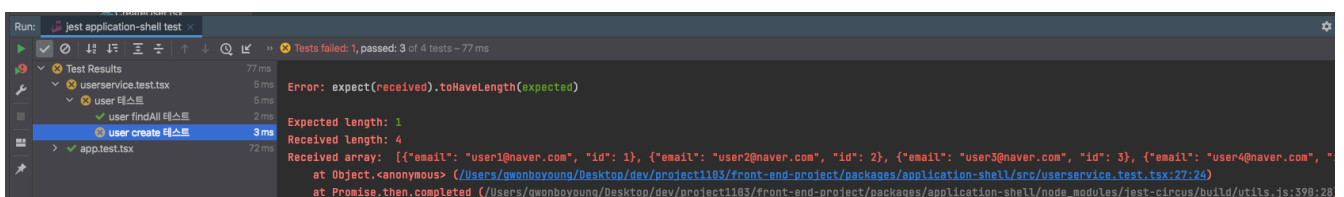
    userService.create(user);

    expect(data.users).toHaveLength(1);
  });
});
```

user create . userService create() .

한명의 유저를 추가하였기에 toHaveLength는 1을 리턴한다고 생각하지만, 이 테스트 코드는 실패합니다.

어떤



user 배열에는 이미 이전의 'user findAll 테스트' 에서 추가된 데이터들이 두번째 테스트에 영향을 끼치기 때문입니다.

각 테스트가 서로 연관없이 동작하게 하고싶다면, 위와 같은 상황을 피하기 위해서는 테스트 실행 후 data 모듈에 저장되어있는 데이터를 정리해주는 작업이 필요합니다.

```

const userService = require("../userService");
const data = require("../data");

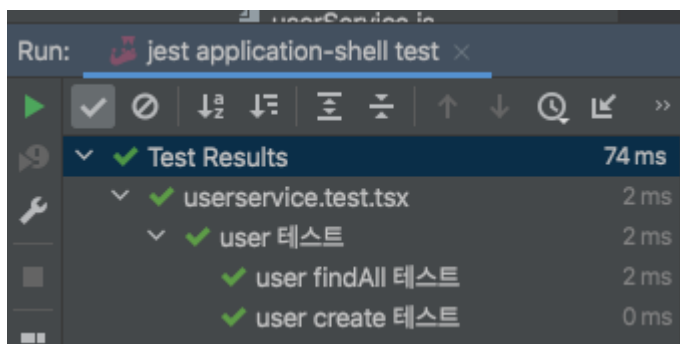
describe('user ', () => {
  it("user findAll ", () => {
    //
  });

  it("user create ", () => {
    //
  });

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

```

afterEach      data.user



각 테스트가 통과하게 됩니다.

```

const userService = require("./userService");
const data = require("./data");

describe('user ', () => {
  it("user findAll ", () => {
    //data users 3
    data.users.push(
      { id: 1, email: "user1@naver.com" },
      { id: 2, email: "user2@naver.com" },
      { id: 3, email: "user3@naver.com" }
    );

    // userService.findAll()
    const users = userService.findAll();

    expect(users).toHaveLength(3);
    expect(users).toContainEqual({ id: 1, email: "user1@naver.com" });
    expect(users).toContainEqual({ id: 2, email: "user2@naver.com" });
    expect(users).toContainEqual({ id: 3, email: "user3@naver.com" });
  });

  it("user delete ", () => {
    data.users.push(
      { id: 1, email: "user1@naver.com" },
      { id: 2, email: "user2@naver.com" },
      { id: 3, email: "user3@naver.com" }
    );

    userService.delete(3)
    expect(data.users).toHaveLength(2);

  });

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

```

이번에는 userService의 delete 함수를 테스트하기 위한 테스트 코드를 추가하였습니다.

delete 함수를 위해 data.users에 데이터를 넣어주고 delete 함수를 실행하면 테스트 코드는 문제없이 돌아가지만,

```

user findAll
.

```

이럴 때는 초기 데이터 적재하는 코드를 beforeEach에 써주어 각 테스트가 실행할 때마다 데이터를 적재할 수 있게 해주어 코드를 간결하게 만들 수 있습니다.

```

const userService = require("../userService");
const data = require("../data");

describe('user ', () => {

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

  it("user findAll ", () => {

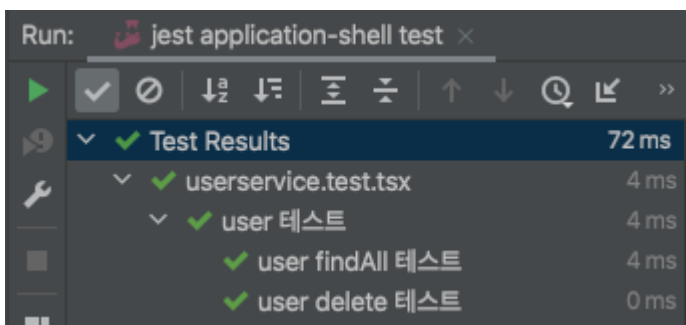
    const users = userService.findAll();

    expect(users).toHaveLength(3);
    expect(users).toContainEqual({ id: 1, email: "user1@naver.com" });
    expect(users).toContainEqual({ id: 2, email: "user2@naver.com" });
    expect(users).toContainEqual({ id: 3, email: "user3@naver.com" });
  });

  it("user delete ", () => {
    userService.delete(3)
    expect(data.users).toHaveLength(2);
  });

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

```



코드가 훨씬 간결해졌으며, 테스트 또한 성공한 것을 볼 수 있습니다.

이번 예시처럼 각 테스트마다 데이터를 적재해줘야하고 테스트마다 서로 영향을 받지 않아야하는 상황과 유사하다면 `beforeEach`와 `afterEach`를 통해 테스트 코드를 관리하고

한번만 데이터 적재 후 각 테스트의 행동이 서로 영향을 끼쳐도 상관없는 상황이라면 `beforeAll`과 `afterAll`을 사용하여 테스트 코드를 관리합니다.

## 3.1.2 Testing-Library

render 함수를 실행하면 그 안에는 다양한 쿼리 함수들이 있습니다.

이 쿼리 함수들은 react-testing-library의 기반인 dom-element-library에서 지원하는 함수들입니다.

쿼리에는 다양한 종류가 있으며 testing-library 공식 문서에 들어가면 다양한 쿼리 함수 소개를 볼 수 있습니다.

### 3.1.2.1 쿼리

getBy\*: getBy\*로 시작하는 쿼리는 조건에 일치하는 DOM 엘리먼트 하나를 선택합니다. 만약 없으면 에러가 발생합니다.

getAllBy\*: getAllBy\* 로 시작하는 쿼리는 조건에 일치하는 DOM 엘리먼트 여러개를 선택합니다. 만약에 하나도 없으면 에러가 발생합니다.

queryBy\*: queryBy\* 로 시작하는 쿼리는 조건에 일치하는 DOM 엘리먼트 하나를 선택합니다. 만약에 존재하지 않아도 에러가 발생하지 않습니다.

queryAllBy\*: queryAllBy\* 로 시작하는 쿼리는 조건에 일치하는 DOM 엘리먼트 여러개를 선택합니다. 만약에 존재하지 않아도 에러가 발생하지 않습니다.

findBy\*: findBy\* 로 시작하는 쿼리는 조건에 일치하는 DOM 엘리먼트 하나가 나타날 때까지 기다렸다가 해당 DOM 을 선택하는 Promise를 반환합니다. 기본 timeout인 4500ms 이후에도 나타나지 않으면 에러가 발생합니다. → 비동기 테스트에 쓰일 수 있습니다.

findAllBy\*: findBy\* 로 시작하는 쿼리는 조건에 일치하는 DOM 엘리먼트 여러개가 나타날 때까지 기다렸다가 해당 DOM 을 선택하는 Promise를 반환합니다. 기본 timeout인 4500ms 이후에도 나타나지 않으면 에러가 발생합니다. → 비동기 테스트에 쓰일 수 있습니다.

- \*에 오는 쿼리

ByLabelText: ByLabelText 는 label 이 있는 input 의 label 내용으로 input 을 선택합니다.

ByPlaceholderText: ByPlaceholderText 는 placeholder 값으로 input 및 textarea 를 선택합니다.

ByText: ByText는 엘리먼트가 가지고 있는 텍스트 값으로 DOM 을 선택합니다.

ByAltText: ByAltText 는 alt 속성을 가지고 있는 엘리먼트 (주로 img) 를 선택합니다.

ByTitle: ByTitle 은 title 속성을 가지고 있는 DOM 혹은 title 엘리먼트를 지니고있는 SVG 를 선택 할 때 사용합니다.

ByDisplayValue: ByDisplayValue 는 input, textarea, select 가 지니고 있는 현재 값을 가지고 엘리먼트를 선택합니다.

ByRole: ByRole은 특정 role 값을 지니고 있는 엘리먼트를 선택합니다.

ByTestId ByTestId 는 다른 방법으로 못 선택할때 사용하는 방법입니다. 특정 DOM 에 직접 test 할 때 사용할 id 를 달아서 선택하는 것을 의미합니다.

### 3.1.2.2 쿼리 예시



```
import React from 'react';

const App: React.FC = params => {

  return (
    <>
      <div> </div>

      <label htmlFor="input-product"></label>
      <input type="text" id="input-product"/>

      <label htmlFor="input-delivery"></label>
      <input type="text" id="input-delivery" placeholder=" " />

      <label htmlFor="input-shipping"></label>
      <input type="text" id="input-shipping" defaultValue='1000' />

      <button></button>

      <div data-testid="input-check"> </div>

    </>
  );
};

export default App;
```

위 쿼리들을 예시로 위 코드를 테스트해보겠습니다.

쿼리를 사용하는 방법은 다양합니다. 아래 코드는 쿼리를 사용하는 세가지 방법을 의사코드로 작성하였습니다.

공식문서는 2번째 방법을 추천하고있습니다.

```

//1.
describe(' ', () => {
  it(' ', () => {
    const { , getByText } = render(<App />);
    const divText = getByText('');
  });
});

//2. testing-library screen . .

describe(' ', () => {
  it(' ', () => {
    render(<App />);
    const divText = screen.getByText('');
    expect(divText).toBeInTheDocument()
  });
});

//3. import container
import {render, screen, getByText} from '@testing-library/react';

describe(' ', () => {

  it(' ', () => {
    const {container} = render(<App />);
    const divText = getByText(container, '') // .
    expect(divText).toBeInTheDocument()
  });
});

```

```
describe(' ', () => {

  it(' ', () => {

    render(<App />); // .

    const divText = screen.getByText(''); // <App> '' .
    expect(divText).toBeInTheDocument() //divText <App> .

    const inputProduct = screen.getByLabelText('');
    expect(inputProduct).toBeInTheDocument()

    const inputDelivery= screen.getByPlaceholderText(' ');
    expect(inputDelivery).toBeInTheDocument()

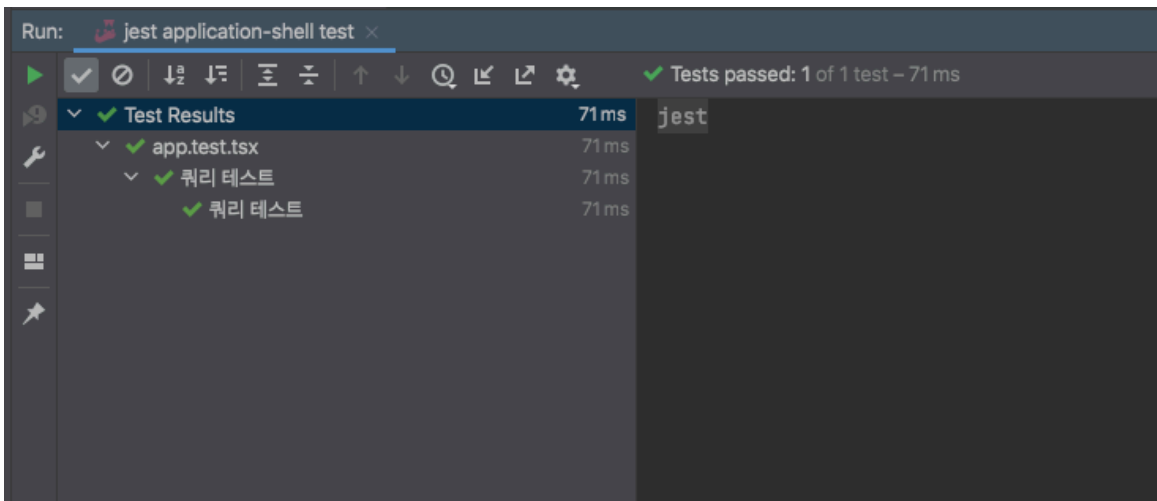
    const inputShipping= screen.getByDisplayValue('1000');
    expect(inputShipping).toBeInTheDocument()

    const buttonRole = screen.getByRole('button', { name: '' });
    expect(buttonRole).toBeInTheDocument();

    const inputCheck = screen.getByTestId('input-check');
    expect(inputCheck).toBeInTheDocument()

  });
});
```

2번째 방법인 testing-library가 제공하는 screen을 이용해 테스트코드를 작성하였습니다. 테스트를 실행하면,



테스트가 성공합니다.

### 3.1.2.3 어떤 쿼리를 써야할까

쿼리의 종류가 다양한 만큼 어떤 쿼리를 써야할지에 대해 고민할 수 있습니다.

testing-library는 공식 문서에서 우선순위에 따라 사용하는 것을 권장하고 있습니다. 아래 내용은 공식 문서(<https://testing-library.com/docs/queries/about/#priority>)를 번역한 내용입니다.

#### 1. 모두가 접근하기 쉬운 쿼리

1. `getByRole`: accessibility tree(접근성 트리)에 보여지는 모든 요소를 조회하기 위해 사용될 수 있습니다. `role`의 `name` 옵션을 사용하면 반환되는 요소를 사용가능한 이름으로 필터링 할 수 있습니다. 이 메소드는 우선순위를 높게 사용해야 합니다.  
→ 사용한다면 `getByRole('button', {name:/submit/})` 처럼 `getByRole`의 `name` 옵션을 사용합니다. `role`이 가지고있는 옵션을 확인해보면 좋습니다.
  2. `getByLabelText`: 이 메소드는 폼에서 사용하기 좋습니다. 웹 사이트 양식을 탐색할 때 사용자는 레이블 텍스트를 사용하여 요소를 찾습니다. 이 메소드 또한 사용을 권장합니다.
  3. `getByPlaceholderText`: label 만큼은 아니지만 다른 대안이 없다면 쓰면 좋습니다.
  4. `getByText`: 폼 양식 외에 텍스트는 사용자가 찾는 주요 방법입니다. 이 메소드는 `div` `span` `p`태그와 같은 비 상호작용 요소를 찾는 데 사용할 수 있습니다
  5. `getByDisplayValue`: 폼 요소의 `value`값은 입력된 값이 있는 페이지를 탐색할 때 유용하게 사용될 수 있습니다
2. **HTML5 과 ARIA에** 부응하는 선택자입니다. 이들 속성을 조작하는 사용방법은 브라우저 및 보조 기술에 따라 크게 다르다는 점에 유의해야 합니다.
1. `getByAltText`: `alt`를 지원하는 `img`, `area`, `input`등의 요소일 경우 사용할 수 있습니다
  2. `getByTitle`: `title` 속성을 가지고 있는 DOM 혹은 `title` 엘리먼트를 지니고있는 SVG 를 선택 할 때 사용합니다. 하지만 `title` 속성은 screen readers에 의해 일관되게 읽히지 않으며 보는 사용자에게 기본적으로 표시되지 않습니다
3. **Test ID**
1. `getByTestId`: 사용자가 볼 수 없거나 `role`이나 텍스트로 매치할 수 없을 때만 권장합니다. → 테스트 코드에 DOM의 `querySelector`를 사용하는 것보단 권장되지만 되도록 사용하지 않습니다.

→ accessibility tree(접근성 트리)란

접근성 트리란 대부분의 HTML 요소에 대한 접근성 관련 정보를 포함하고 있습니다.

브라우저는 마크업을 DOM 트리라는 내부 표현으로 합니다.

DOM 트리는 모든 마크업 요소, 속성 및 텍스트 노드를 나타내는 객체를 포함합니다.

그런 다음 브라우저는 DOM 트리를 기반으로 접근성 트리를 생성하는데, 이 트리는 플랫폼별 접근성 API가 화면 판독기와 같은 보조 기술로 이해할 수 있는 표현을 제공하기 위해 사용합니다.

`getByLabelText`: 이 메소드는 폼에서 사용하기 좋습니다. 웹 사이트 양식을 탐색할 때 사용자는 레이블 텍스트를 사용하여 요소를 찾습니다. 이 메소드 또한 사용을 권장합니다.

[https://developer.mozilla.org/en-US/docs/Glossary/Accessibility\\_tree](https://developer.mozilla.org/en-US/docs/Glossary/Accessibility_tree) 번역

## 3.2 Jest, Testing Library를 이용한 테스트코드 작성

### 3.2.1 버튼 클릭시 글자가 나타나는 컴포넌트 테스트

```
import React, {useState} from 'react';

function App() {
  const [str, setStr] = useState<string>('');

  const handleSetStr = () => {
    if(str == ''){
      setStr(' .');
    }else {
      setStr('');
    }
  }

  return (
    <div>
```

```

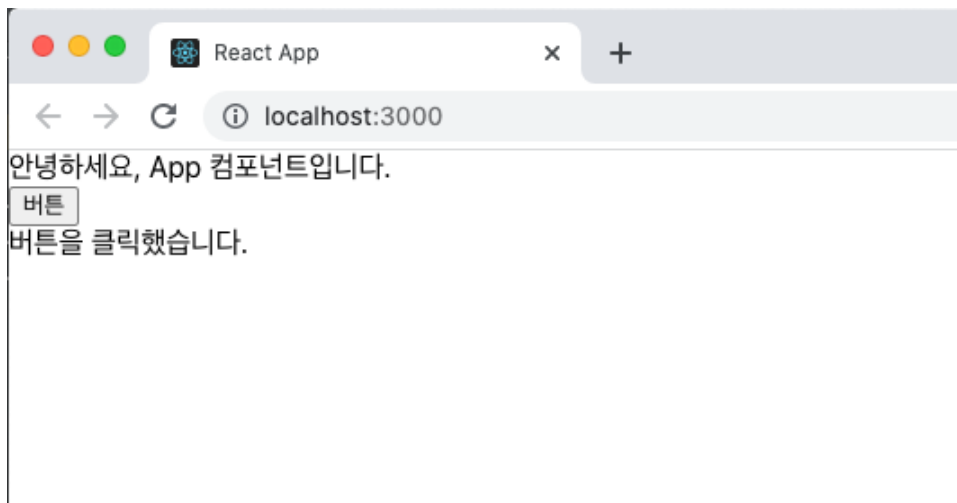
    <div>
      , App .<br/>
      <button onClick={handleSetStr}> </button>
      <div > {str} </div>
    </div>
  </div>
);
}

export default App;

```

위 코드로 이루어진 컴포넌트는 버튼을 누를시 '버튼을 클릭했습니다.' 라는 글자를 띄우는 컴포넌트입니다.

실행시 아래와 같이 실행이 됩니다.



이 컴포넌트를 jest와 testing-library를 통해 테스트해보겠습니다.

```

import React from 'react';
import { render, screen, fireEvent } from '@testing-library/react';
import App from './app';

describe('App test', () => {

  it('App - ', () => {
    render(<App />); // App ,
    const appBtn = screen.getByText(''); // '' DOM .

    fireEvent.click(appBtn); // appBtn .

    const appStr = screen.getByText(' '); // ' ', ''
    .

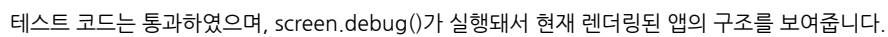
    expect(appStr).toBeInTheDocument(); // ' ' appStr
    .

    screen.debug(); // console .
  });

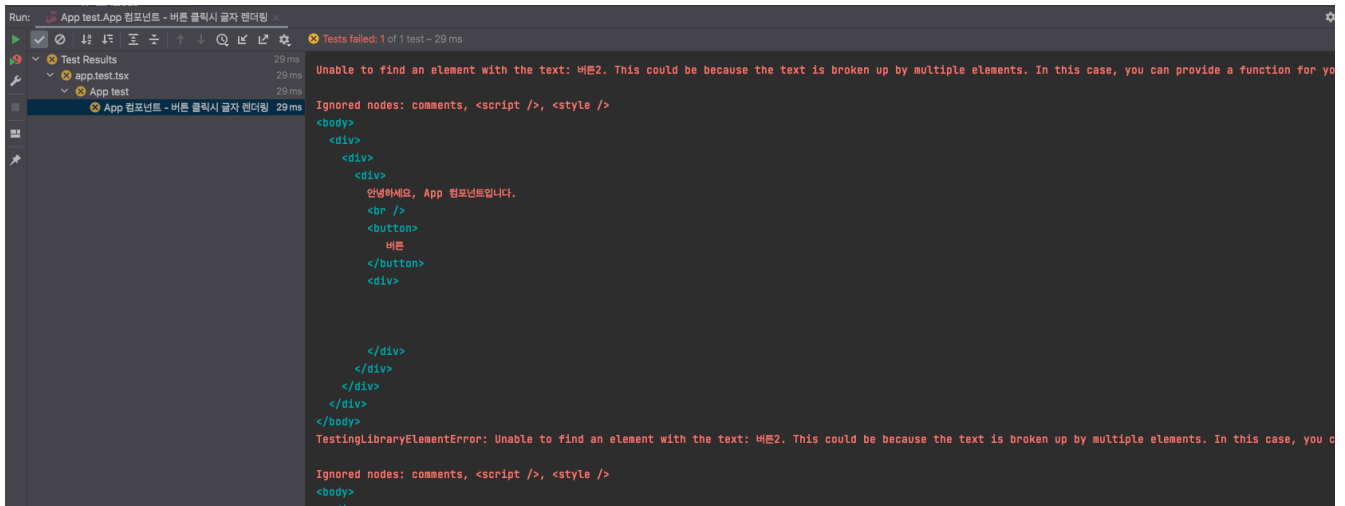
});

```

테스트 코드를 실행하게 되면



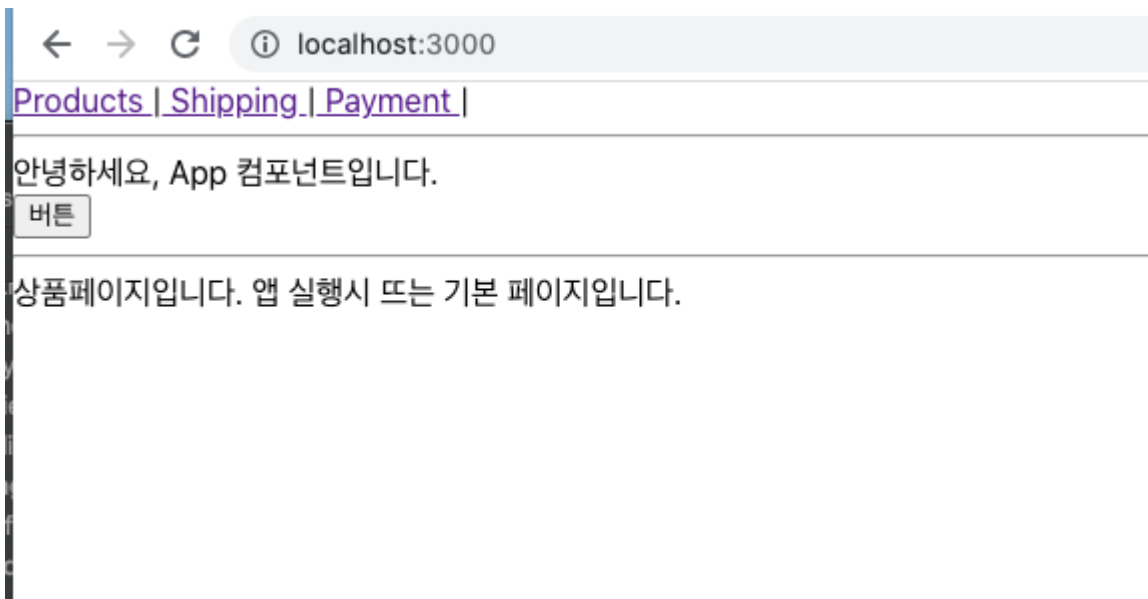
위 코드처럼 존재하지 않는 버튼2 텍스트를 찾는 테스트 코드를 실행 하면



'버튼2' 텍스트를 가진 엘리먼트를 찾을 수 없다는 오류메세지와 현재 렌더된 컴포넌트의 구조를 보여주며 테스트가 실패하게 됩니다.

### 3.2.2 React Lazy, React Router로 이루어진 SPA 컴포넌트 테스트

저희가 개발 중인 프론트엔드 환경설정에서 쓰이는 동적 임포트를 도와주는 React Lazy와 React Router로 이루어진 컴포넌트를 테스트해보겠습니다.



상단의 메뉴들인 Products, Shipping, Payment를 누르면 각 주소에 맞는 컴포넌트가 제일 하단의 div에 보이게 됩니다.

App.tsx

```
import React, {Suspense, useState} from 'react';
import {BrowserRouter as Router, Link, Route, Switch} from 'react-router-dom';
import Loading from './loading';

const App: React.FC = params => {
  const [str, setStr] = useState<string>('');

  const Products = React.lazy(()=>import('./component/products'));
  const Shipping = React.lazy(()=>import('./component/shipping'));
```

```

const Payment = React.lazy(()=>import('./component/payment'));

//
return (
  <Router>
    <div>
      <nav>
        <Link to="/"> Products </Link>
        <Link to="/shipping" data-testid="shipping"> Shipping </Link>
        <Link to="/payment"> Payment </Link>
      </nav>

      //

      <div className="app-shell-main-content">
        <Suspense fallback={<Loading />}>
          <Switch>
            <Route exact path="/">
              <Products/>
            </Route>

            <Route path="/shipping">
              <Shipping/>
            </Route>

            <Route path="/payment">
              <Payment/>
            </Route>
          </Switch>
        </Suspense>
      </div>

    </div>
  </Router>
);
};

export default App;

```

간단한 컴포넌트 예제에서 추가된 코드입니다.

```

import React, { Suspense } from 'react';
import { render, screen, waitFor } from '@testing-library/react';
import App from './app';

describe('App test', () => {

  //

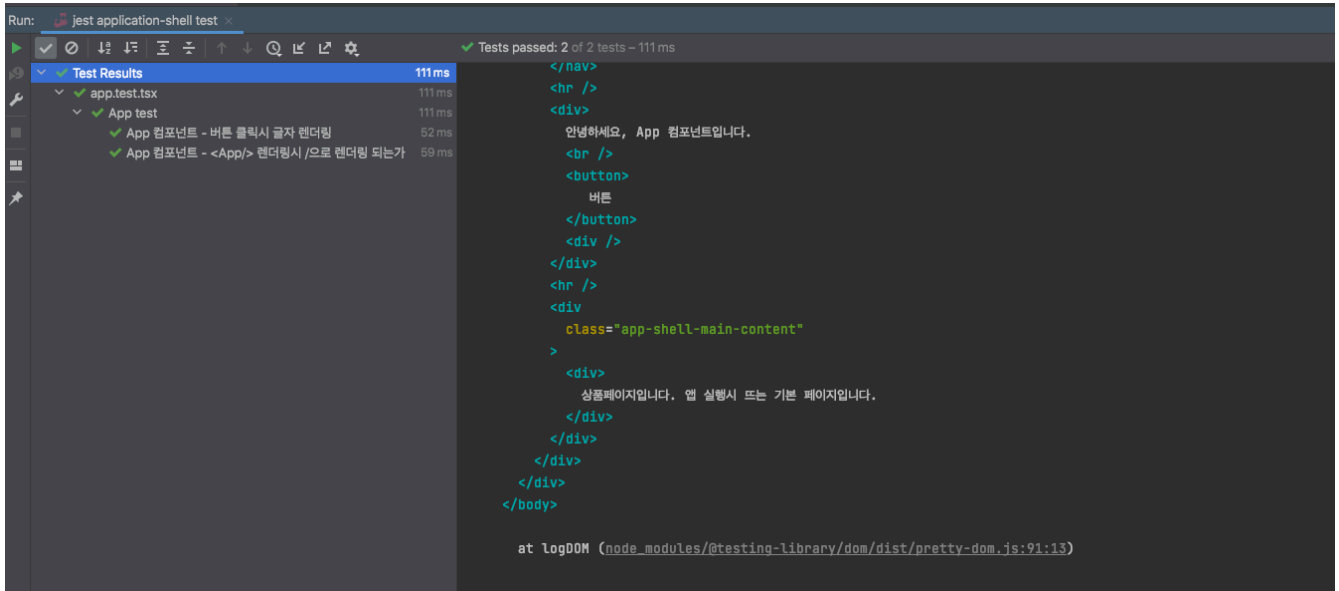
  it('App - <App/> / ', async () => {
    render(<App/>);
    await waitFor(() => expect(screen.getByText(' . ')).
    toBeInTheDocument());
  });

```



```
});
});
```

간단한 컴포넌트 테스트 예제에서 추가된 코드입니다.



테스트 코드를 실행하면 실행시 주소 <http://localhost:3010/> 에 맞는 <Products> 컴포넌트가 <div class="app-shell-main-content">에 렌더된 것을 볼 수 있으며 테스트가 성공하였습니다.

```
describe('App test', () => {

  //

  it('App - <App/> / ', async () => {
    render(<App />);
    await waitFor(() => expect(screen.getByText(' . ')).
    toBeInTheDocument())
    screen.debug();
  });

  it('App - Shipping <Shipping> ', async () => {
    render(<App />);

    const shippingMenu = screen.getByTestId('shipping'); //data-testid
    .
    fireEvent.click(shippingMenu)
    await waitFor(() => expect(screen.getByText(/./i)).
    toBeInTheDocument()) //text .

  });

  it('App - Payment <Payment> ', async () => {
    render(<App />);

    const paymentMenu = screen.getByText('Payment');
    fireEvent.click(paymentMenu)
```

```

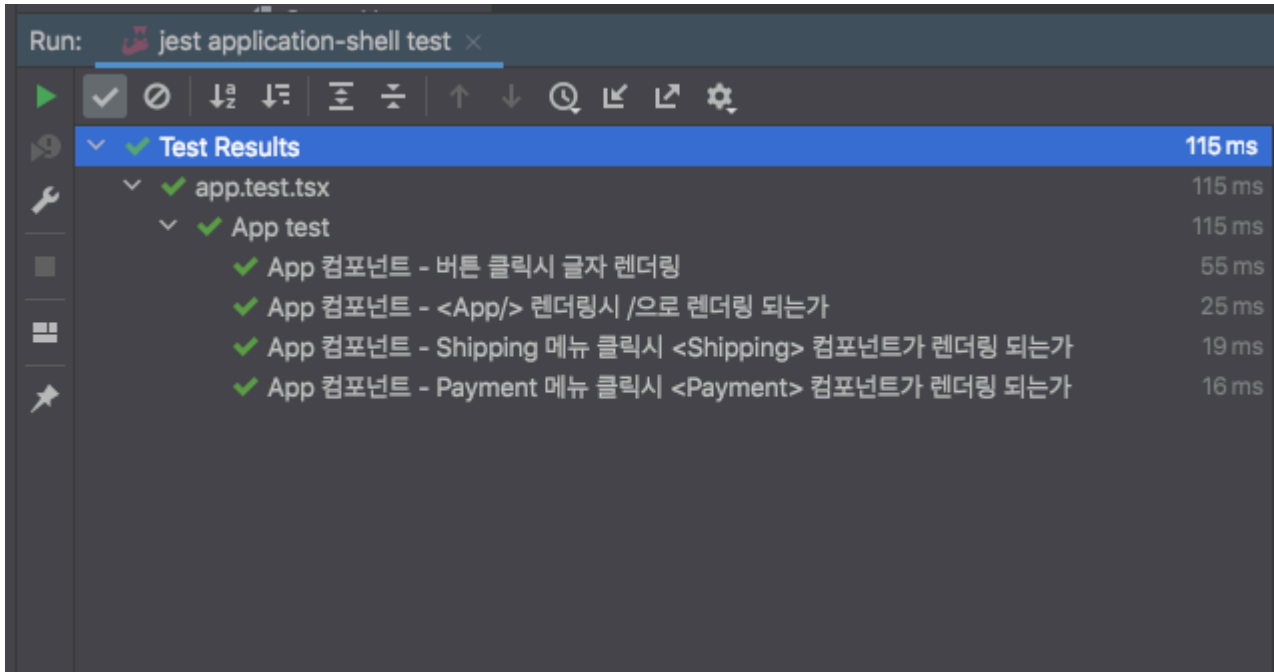
    await waitFor(() => expect(screen.getByText(' .')).
    toBeInTheDocument())

  });

});

```

최종적으로 각 메뉴 클릭시 그 주소에 맞는 컴포넌트가 렌더링되는지 테스트하는 코드입니다. 이를 실행하면



테스트 코드가 성공했음을 볼 수 있습니다.

### 3.2.2.1 Testing-Library Async Utilities

Testing-Library에서는 비동기 코드를 처리하기 위해 Async Utilities에서 함수가 제공됩니다.

```

import React, {Suspense, useState, useCallback} from 'react';

const App: React.FC = params => {
  const [toggle, setToggle] = useState(false);

  const onToggle = useCallback(() => {
    setTimeout(() => {
      setToggle(toggle => !toggle);
    }, 1000);
  }, []);

  return (
    <div>
      <button onClick={onToggle}> </button>
      <div>
        : <span> {toggle ? 'ON' : 'OFF'} </span>
      </div>

      {toggle? <div> on </div> : null}
    </div>
  );
};

```

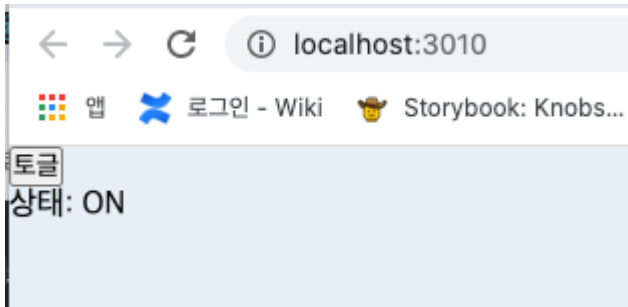
```

    </div>
  );
};

export default App;

```

위 코드는 버튼 클릭시 1초 뒤에 toggle의 상태가 true→ false | false → true로 바뀌는 버튼입니다.



실행 화면입니다. 버튼 클릭시 1초 후에 상태:OFF로 바뀌게됩니다. 이 컴포넌트를 Async Utilities의 함수를 통해 테스트해보겠습니다.

```

import {render, waitFor, waitForElementToBeRemoved, fireEvent} from
 '@testing-library/react';

describe('App test', () => {
  it('toggle ', async() => {
    render(<App/>);
    const toggleButton = screen.getByText('');

    fireEvent.click(toggleButton);
    await waitFor(() => screen.getByText(' on '));
  })
})

```

**wait** → 버전이 올라가면서 **waitFor**로 바뀌었습니다.

특정 콜백에서 에러가 발생하지 않을 때까지 기다리다가 대기시간이 timeout을 초과하게 되면 테스트케이스가 실패합니다.

timeout은 기본값 4500ms이며 timeout 시간을 조절할 수 있습니다.

위 3.2.2 테스트 예시에서는 각 주소에 맞는 컴포넌트가 뜨기 전 <Loading/> 컴포넌트를 띄우는데 각 주소에 맞는 컴포넌트가 뜰 때까지 기다리게 사  
용했습니다.

```

import {render, waitFor, waitForElementToBeRemoved, fireEvent} from
 '@testing-library/react';

describe('App test', () => {
  //

  it('toggle - waitForElementToBeRemoved', async () => {
    render(<App />);
    const toggleButton = screen.getByText('');

```

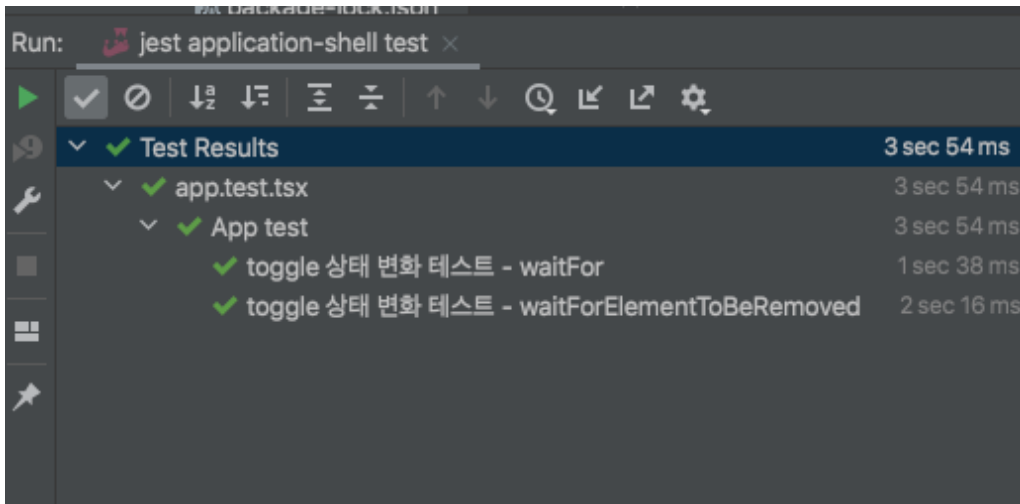
```

    fireEvent.click(toggleButton);
    const onText = await waitFor( () => screen.getByText(' on '));

    fireEvent.click(toggleButton);
    await waitForElementToBeRemoved(onText); //onText
  });
})

```

**waitForElementToBeRemoved** → 특정 엘리먼트가 화면에서 사라질때까지 기다리는 함수입니다.



두개의 테스트 코드가 성공하였습니다.

### 3.2.3 Mock함수를 이용한 테스트코드 작성

Mock함수는 실제 함수를 구현한 것이 아니라 흉내만 낸, 테스트하기 위해서 만든 일종의 모형이라고 할 수 있습니다. Mock함수는 다양하게 사용될 수 있습니다.

- user를 실제 DB에 생성하는 함수를 테스트할 경우, 테스트할 때마다 DB에 유저가 생성되고 이를 지우기위해 Rollback을 매일 해준다면 번거로운 것입니다. 이를 해결하기 위해 mocking module이라는 것을 사용하여 테스트할 수도 있습니다.

- 또한 외부 DB, 서버와 연결된 함수인 경우 외부 요인의 영향에 따라 테스트또한 달라질 수 있습니다. 테스트에서는 동일한 코드는 동일한 결과를 내는 것이 중요하기 때문에 이런 경우에도 mock 함수를 사용할 수 있습니다.

#### 3.2.3.1 mock함수 사용하기

```

describe('Mock test', () => {

    const mockFn = jest.fn(); //mock

    mockFn(); //mock
    mockFn(1); //mock 1

    it('mock ', ()=>{
        console.log(mockFn.mock.calls); //mockFn mock.calls

        expect(mockFn.mock.calls.length).toBe(2); //
        expect(mockFn.mock.calls[1][0]).toBe(1); //
    })

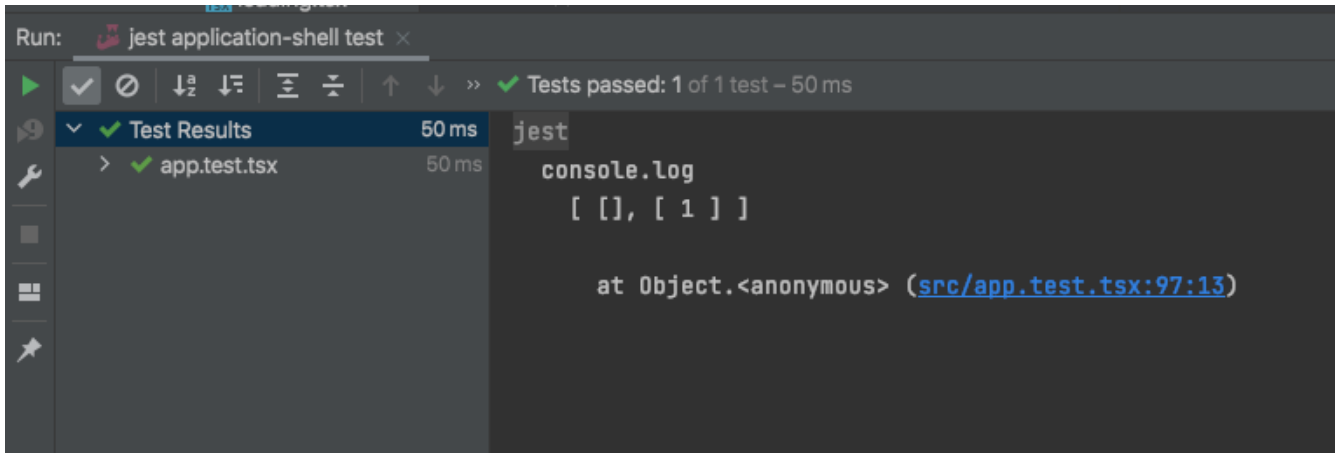
```

```

        expect(mockFn).toBeCalledTimes(2); //mock
        expect(mockFn).toBeCalledWith(1); //mock
    });
});

```

위 코드는 mock 함수 사용 예시입니다. 다음 테스트를 실행하면



배열이 리턴되는 것을 볼 수 있습니다. 배열의 길이는 함수가 호출된 수이고, 배열안에 들어있는 것은 mock 함수에 전달된 인수입니다.

```

describe('Mock test', () => {
    const mockFn = jest.fn();

    mockFn
        .mockReturnValueOnce(true)
        .mockReturnValueOnce(false)
        .mockReturnValueOnce(true)
        .mockReturnValueOnce(false)
        .mockReturnValueOnce(true)

    const result=[1,2,3,4,5].filter(num => mockFn(num));

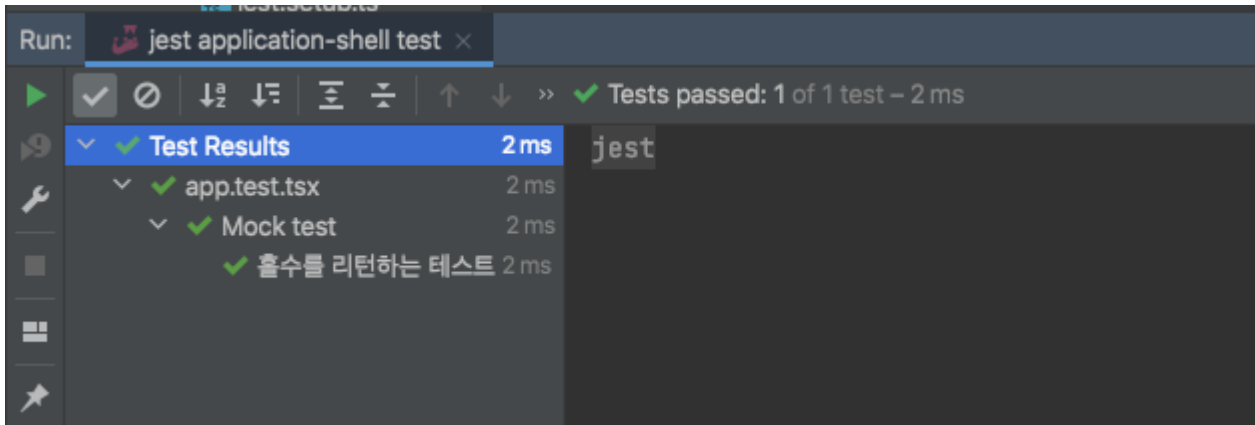
    it(" .", () => {
        expect(result).toStrictEqual([1,3,5]);
    });
});

```

mockReturnValue(리턴값) 함수를 이용해서 mock함수가 어떤 값을 리턴해야할지 설정해줄 수 있습니다.

위 테스트코드 예시는 홀수를 리턴하는 함수를 mock함수로 구현한 예시입니다.

이는 실제로 홀수만 리턴하는 함수가 아닙니다. 어떤 숫자가 들어오든지 순서대로 true false true false true를 리턴하는 Mock함수입니다.

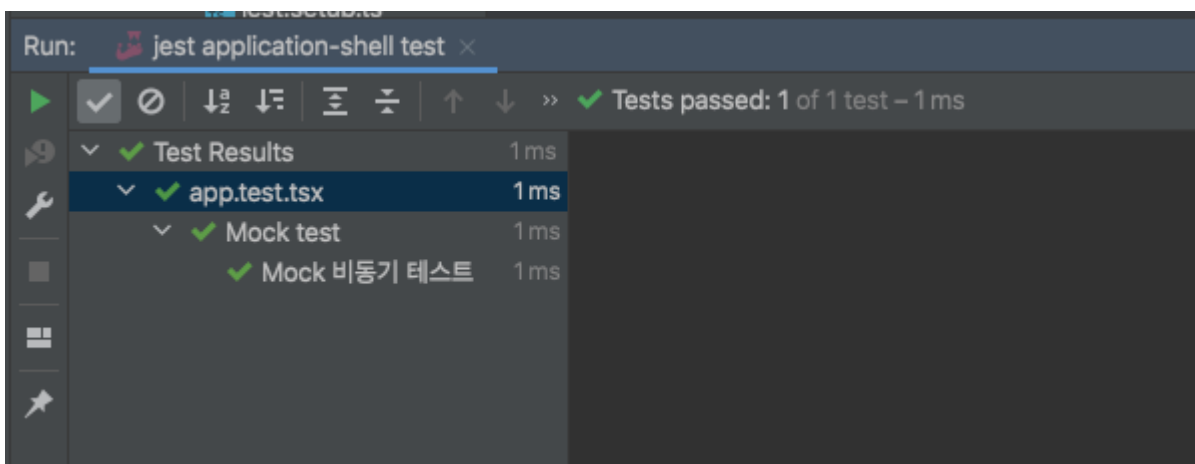


테스트가 성공하였습니다.

```
describe('Mock test', () => {
  const asyncMockFn = jest.fn();
  asyncMockFn.mockResolvedValue({name: "Mike"});

  it('Mock ', () => {
    asyncMockFn().then((res: { name: string; }) => { //
      expect(res.name).toBe("Mike");
    })
  });
});
```

이와 유사한 mockResolvedValue(Promise가 resolve하는 값) 함수를 이용하면 가짜 비동기 함수를 만들어서 실제 비동기 함수처럼 테스트를 할 수 있습니다.



### 3.2.3.2 mocking module

fn.js

```
const fn = {
  createUser: (name) => {
```

```

        console.log(' .')
        return {
            name, // DB user      return .
        }
    }
}

```

test 코드

```

describe('Mock test', () => {

    const fn = require("./fn");

    it('user test', () => {
        const user = fn.createUser("Jun");
        expect(user.name).toBe("Jun");
    })
});

```

위 테스트 코드는 fn.createUser 함수가 "Jun"을 매개변수로 넘겨줬을 때 DB에 생성 후 생성한 유저를 return하는지 테스트하는 함수입니다.

이 코드는 테스트를 할 때마다 DB에 Jun이라는 유저가 생성이 될 것이며 이는 테스트 할 때마다 Rollback을 해줘야하는 불편함이 있습니다.

```

describe('Mock test', () => {

    const fn = require("./fn");
    jest.mock("./fn"); //mock   fn mocking module . fn   mock .
    fn.createUser.mockReturnValue({name:"Jun"}); //   fn.createUser   Jun
    mock .

    it('user test', () => {
        const user = fn.createUser("Jun");
        expect(user.name).toBe("Jun");
    })
});

```

mocking module을 이용하면 실제 user 생성 없이 테스트코드를 동작할 수 있습니다.

위와 유사한 예시인 아래의 두 코드는 프론트엔드 미니프로젝트에서 사용했던 예시입니다.

```

import axios from "axios";

export async function GetProduct(){
    return await axios.get("http://localhost:3030/GetProduct");
}

```

axios.get로 데이터를 받아오는 코드를 테스트해보겠습니다.

```
import {GetProduct} from "../GetProduct";
import axios from "axios";

jest.mock('axios'); // . axios Mock.

describe(' ', () => {
  it('axios get ', async () => {
    (axios.get as jest.Mock).mockReturnValue({ //Mock axios.get
      mockReturnValue: {
        BrandName: '',
        ProductName: ' a-123',
      }
    });

    const result = await GetProduct(); //GetProduct axios.get Mock .
    expect(result).toEqual({
      BrandName: '',
      ProductName: ' a-123',
    });

    expect(axios.get).toHaveBeenCalledTimes(1);
    expect(axios.get).toHaveBeenCalledWith('http://localhost:3030
/GetProduct');
  });
});
```

jest.fn -> 하나의 함수를 mock 처리합니다.

jest.mock -> 모듈을 인수로 받을 수 있으며 모듈내의 함수 전부를 mock처리합니다.

위 테스트 코드는 실제로 DB에서 값을 가져오는게 아니라 axios모듈을 mock처리해서 어떤 인수가 들어와도 mockReturnValue로 정한 값을 리턴하게 합니다.

이처럼 다양하게 mock으로 테스트를 할 수 있습니다.