리액트 테스트 라이브러리

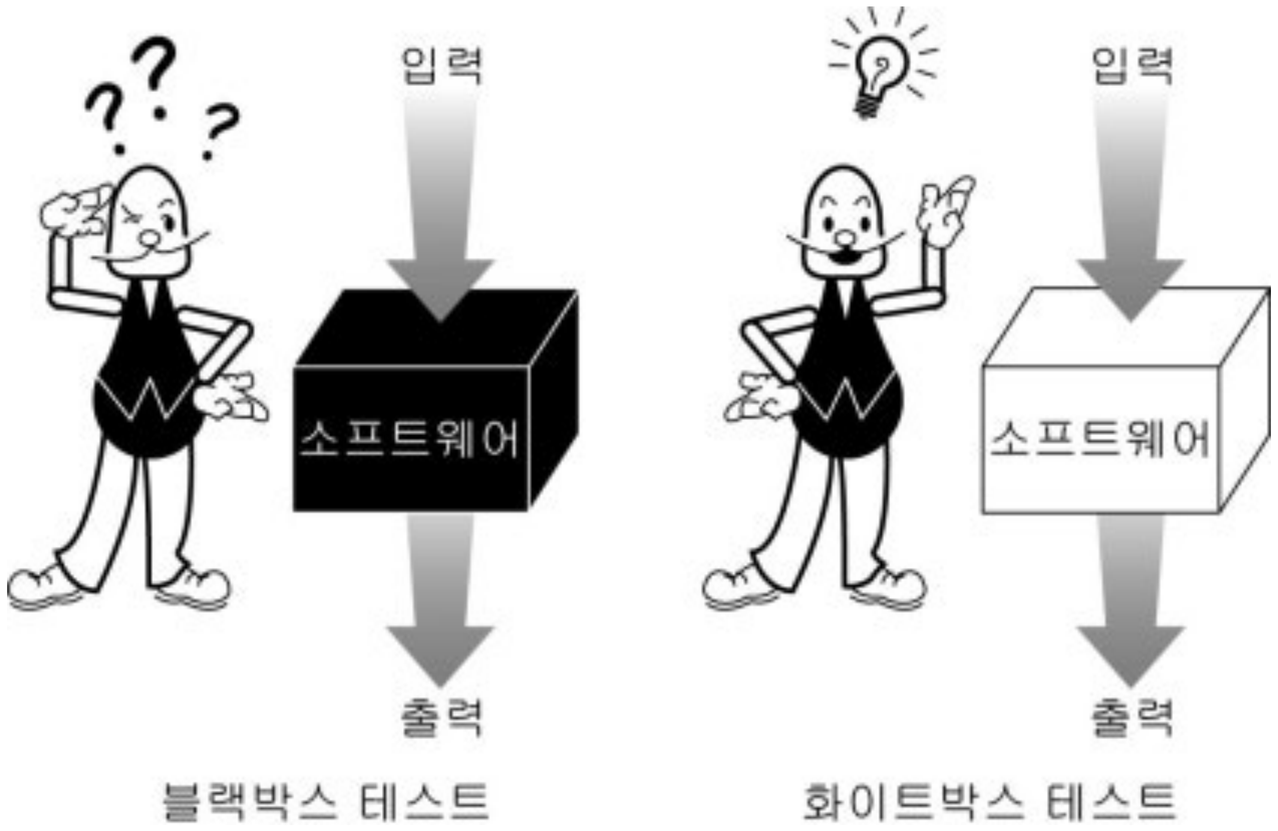
2025.02.03
정상영

프로그래밍에서의 테스트 ?

테스트의 기대효과

1. 설계한 대로 로직이 정확히 동작하는지 확인
2. 버그를 사전에 발견하고 방지
3. 오작동으로 인한 비용과 리소스 낭비 감소
4. 코드의 유지보수성과 확장성 향상

프론트엔드 테스트 vs 백엔드 테스트



백엔드의 테스트

1. 화이트박스 테스트
2. 작성한 로직을 전부 이해하고 있어야 함
3. 주로 어플리케이션에서 수행

프론트엔드의 테스트

1. 블랙박스 테스트
2. 로직보다 의도대로 동작하는지가 중요
3. 사용자와 비슷한 환경에서 수행

프론트엔드의 테스트

1. 유닛 테스트: 각 코드나 컴포넌트가 독립적으로 분리된 환경에서 의도대로 동작하는지 테스트
2. 통합 테스트: 유닛 테스트를 통과한 여러 컴포넌트가 하나로 묶여서 동작하는지 테스트
3. 엔드 투 엔드(E2E) 테스트: 실제 사용자처럼 작동하는 로봇을 활용해 애플리케이션의 전체 동작을 확인하는 테스트

Node.js의 assert

```
const assert = require('assert')

function sum(a, b) {
  return a + b
}

assert.equal(sum(1, 2), 3)
assert.equal(sum(2, 2), 4)
assert.equal(sum(1, 2), 4) // AssertionError [ERR_ASSERTION] [ERR_ASSERTION]: 3 == 4
```

어설션 라이브러리

1. assert처럼 테스트 결과를 확인할 수 있도록 도와주는 라이브러리를 어설션(assertion) 라이브러리라고 함
2. equal, deepEqual, notEqual, throws 등 다양한 메서드 제공
3. should.js, expect.js, chai 등 다양한 라이브러리가 있음

7 Top React Testing Libraries Everyone Should Know

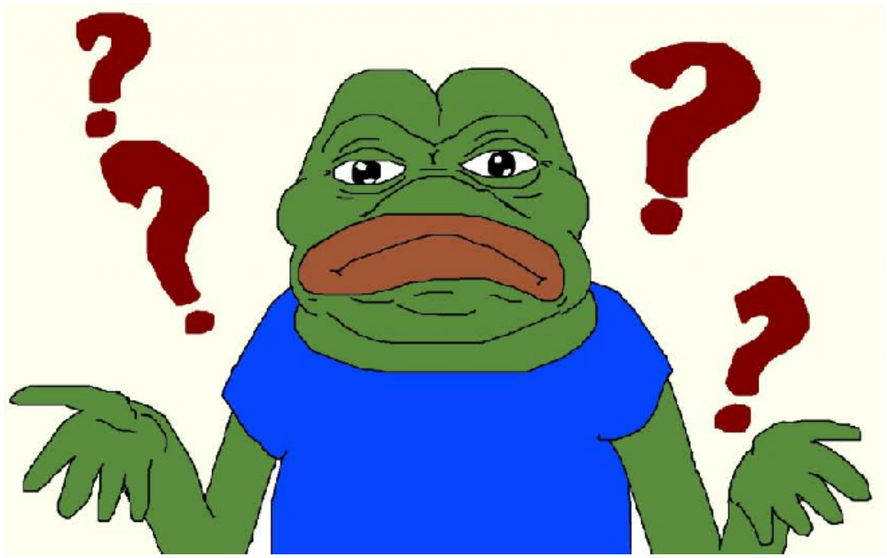
atatus

🐙 React Testing Library (RTL)

↳ dom-testing-library 기반의 리액트 테스트 라이브러리

↳ jsdom 기반의 라이브러리

↳



jsdom

↳ HTML 없이 JavaScript만 존재하는 환경에서
HTML과 DOM을 사용할 수 있도록 해주는 라이브러리

↳ dom-testing-library 는 자바스크립트 환경에서
DOM 조작 가능

↳ 같은 원리로 RTL은 리액트 코드 만으로
브라우저 렌더링 없이 테스트 가능

RTL의 기본 쿼리 함수

1. getBy...: 인수의 조건에 맞는 요소를 반환.
   없거나 두 개 이상이면 에러 발생. (복수는 getAllBy... 사용)

2. findBy...: getBy와 유사하나 Promise를 반환
   비동기 액션 이후에 찾을 때 사용. (복수는 findAllBy...)

3. queryBy...: 조건에 맞는 요소를 반환하지만, 없어도 에러 발생 X
   (복수는 queryAllBy...)

테스팅 프레임워크

1. 테스트 코드가 테스트를 통과하는 것을 넘어
   어떤 테스트가 무엇을 테스트하는지 보고 싶음

2. 코드 작성자에게 소요 시간, 세부 결과, 전체 결과 등
   다양한 정보를 함께 제공

3. Jest, Mocha, Karma, Jasmine 등이 있음

Jest?

1. Facebook에서 만든 All-in-one 자바스크립트 테스팅 프레임워크
2. 테스트할파일명.test.tsx 또는 .jsx
3. npm test로 테스트 실행
4. 어설션에 그치지 않고 Mocking, 스냅샷 테스트, 비동기 처리 등 종합적인 테스트가 가능

Jest의 기본 구조와 개념

1. describe(desc, func): test를 그룹화하고 환경 설정하는데 사용
2. test(content, func): 테스트 케이스를 정의, expect와 함께 사용하여 결과를 검증할 수 있음
3. expect(value).Matchers(result): 테스트 결과를 검증하는 데 사용, result에 예상 값을 넣고 실제 값인 value와 Matcher로 비교한다.
4. matcher: 값과 예상 결과를 확인하는데 사용
   toBe, toEqual, toMatch, toBeDefined 등등 많은 함수가 있음

RTL과 Jest로 리액트 컴포넌트 테스트 해보기

```tsx
//App.test.tsx

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

test('renders learn react link', () => {
  render(<App />); // App 컴포넌트를 렌더링합니다.

  // "learn react"라는 텍스트를 가진 DOM 요소를 찾습니다.
  const linkElement = screen.getByText(/learn react/i);

  // 해당 요소가 DOM에 존재하는지 검증합니다.
  expect(linkElement).toBeInTheDocument();
});
```

- render: 컴포넌트를 가상 DOM에 렌더링
- screen: 렌더링된 DOM에 접근할 수 있도록 쿼리 함수를 제공하는 전역 객체

RTL과 Jest로 리액트 컴포넌트 테스트 해보기

```
// 성공
PASS  src/App.test.js
 ✓ renders learn react link (12 ms)


// 실패
FAIL  src/App.test.js
 × renders learn react link (14 ms)

  ● renders learn react link

    Unable to find an element with the text: /learn react/i.
```

정적 컴포넌트 테스트

```
import { render, screen } from '@testing-library/react'

import StaticComponent from './index'

beforeEach(() => {
  render(<StaticComponent />)
})

describe('링크 확인', () => {
  it('링크가 3개 존재한다.', () => {
    const ul = screen.getByTestId('ul')
    expect(ul.children.length).toBe(3)
  })

  it('링크 목록의 스타일이 square다.', () => {
    const ul = screen.getByTestId('ul')
    expect(ul).toHaveStyle('list-style-type: square;')
  })
})
```

- beforeEach: 각 test를 수행하기 전 실행하는 함수, 전처리기
- it: test 함수의 alias

```jsx
export default function StaticComponent() {
  return (
    <>
      <h1>Static Component</h1>
      <div>유용한 링크</div>

      <ul data-testid="ul" style={{ listStyleType: 'square' }}>
        <li>
          <AnchorTagComponent
            targetBlank
            name="리액트"
            href="https://reactjs.org"
          />
        </li>
        <li>
          <AnchorTagComponent
            targetBlank
            name="네이버"
            href="https://www.naver.com"
          />
        </li>
        <li>
          <AnchorTagComponent name="블로그" href="https://yceffort.kr" />
        </li>
      </ul>
    </>
  )
}
```

동적 컴포넌트 테스트

```tsx
import { fireEvent, render } from '@testing-library/react'
import userEvent from '@testing-library/user-event'

import { InputComponent } from '.'

describe('InputComponent 테스트', () => {
  const setup = () => {
    const screen = render(<InputComponent />)
    const input = screen.getByLabelText('input') as HTMLInputElement
    const button = screen.getByText(/제출하기/i) as HTMLButtonElement
    return {
      input,
      button,
      ...screen,
    }
  }

  it('input의 초기값은 빈 문자열이다.', () => {
    const { input } = setup()
    expect(input.value).toEqual('')
  })
```

- setup: 가상 DOM에 렌더링된 컴포넌트를 재사용할 수 있게 변수화하는 함수

동적 컴포넌트 테스트

```
it('영문과 숫자만 입력된다.', () => {
    const { input } = setup()
    const inputValue = '안녕하세요123'
    userEvent.type(input, inputValue)
    expect(input.value).toEqual('123')
  })

it('버튼을 클릭하면 alert가 해당 아이디로 뜬다.', () => {
    const alertMock = jest
      .spyOn(window, 'alert')
      .mockImplementation((_: string) => undefined)

    const { button, input } = setup()
    const inputValue = 'helloworld'

    userEvent.type(input, inputValue)
    fireEvent.click(button)

    expect(alertMock).toHaveBeenCalledTimes(1)
    expect(alertMock).toHaveBeenCalledWith(inputValue)
  })
})
```

- userEvent: 사용자의 행동을 흉내낼 수 있도록 함
  .type() 메서드로 사용자가 키보드에 타이핑하는 것을 흉내낸 것
- fireEvent: userEvent 보다 섬세한 행동
  userEvent.click을 동작하면 내부적으로 다음과 같이 실행된다.

  1. fireEvent.mouseOver
  2. fireEvent.mouseMove
  3. fireEvent.mouseDown
  4. fireEvent.mouseUp
  5. fireEvent.mouseClick
- jest.spyOn: 소스코드의 동작에 영향을 주지않고 관찰만 할 수 있다.
- jest.spyOn.mockImplementation
  : 모킹(Mocking) 구현에 사용
  현재 코드는 Node.js에서 실행되었기 때문에 window 전역 객체가 존재하지 않는데, 이 메서드를 사용하면 모의의 window를 구현할 수 있다.

  좌측 코드를 정리하면 window.alert에 스파이를 심고 이를 모의 구현한 alertMock이 '몇 번 호출됐는지'와 'helloworld'로 호출되었는지 테스트하는 코드이다.

비동기 이벤트가 발생하는 컴포넌트

```
import { fireEvent, render, screen } from '@testing-library/react'
import { rest } from 'msw'
import { setupServer } from 'msw/node'

import { FetchComponent } from '.'

const MOCK_TODO_RESPONSE = {
  userId: 1,
  id: 1,
  title: 'delectus aut autem',
  completed: false,
}

const server = setupServer(
  rest.get('/todos/:id', (req, res, ctx) => {
    const todoId = req.params.id

    if (Number(todoId)) {
      return res(ctx.json({ ...MOCK_TODO_RESPONSE, id: Number(todoId) }))
    } else {
      return res(ctx.status(404))
    }
  }),
)
```

- msw: Mock Service Worker,
  fetch 요청을 감지하고 준비한 모킹 데이터를
  제공할 수 있는 라이브러리

- setupServer: msw로 서버를 만듬

좌측의 코드를 정리하면
1. api를 모킹한다음
2. todoId가 숫자인지 확인하고
3. 숫자라면 준비한 모킹 response를
4. 아니라면 404 에러를 던진다.

비동기 이벤트가 발생하는 컴포넌트

```
beforeAll(() => server.listen())
afterEach(() => server.resetHandlers());
afterAll(() => server.close())
```

- beforeAll: 테스트 시작 전 서버를 가동
- afterEach: 각 테스트 이후 서버를 기본 설정으로
- afterAll: 테스트 이후 서버 종료

비동기 이벤트가 발생하는 컴포넌트

```
beforeEach(() => {
  render(<FetchComponent />)
})

describe('FetchComponent 테스트', () => {
  it('데이터를 불러오기 전에는 기본 문구가 뜬다.', async () => {
    const nowLoading = screen.getByText(/불러온 데이터가 없습니다./)
    expect(nowLoading).toBeInTheDocument()
  })

  it('버튼을 클릭하고 서버요청에서 에러가 발생하면 에러문구를 노출한다.', async () => {
    server.use(
      rest.get('/todos/:id', (req, res, ctx) => {
        return res(ctx.status(503))
      }),
    )

    const button = screen.getByRole('button', { name: /1번/ })
    fireEvent.click(button)

    const error = await screen.findByText(/에러가 발생했습니다/)
    expect(error).toBeInTheDocument()
  })
})
```

- 좌측의 코드를 정리하면

1. getByText로 가상 DOM에 loading state가 true일 때 노출하는 텍스트를 확인하여 로딩유무를 확인한다.

2. /todos/:id의 모든 response를 503으로 처리하여 에러 상황을 테스트한다.

사용자 정의 훅 테스트

```tsx
export default function useEffectDebugger(
  componentName: string,
  props?: Props,
) {
  const prevProps = useRef<Props | undefined>()

  useEffect(() => {
    if (process.env.NODE_ENV === 'production') {
      return
    }

    const prevPropsCurrent = prevProps.current

    if (prevPropsCurrent !== undefined) {
      const allKeys = Object.keys({ ...prevProps.current, ...props })

      const changedProps: Props = allKeys.reduce<Props>((result, key) => {
        const prevValue = prevPropsCurrent[key]
        const currentValue = props ? props[key] : undefined

        if (!Object.is(prevValue, currentValue)) {
          result[key] = {
            before: prevValue,
            after: currentValue,
          }
        }
        return result
      }, {})

      if (Object.keys(changedProps).length) {
        // eslint-disable-next-line no-console
        console.log(CONSOLE_PREFIX, componentName, changedProps)
      }
    }

    prevProps.current = props
  })
}
```

- 최초 렌더링은 동작하지 않는다.
- 이전 props와 신규 props를 비교해서 무엇이 렌더링을 유발했는지 확인한다.
- 번들러가 프로덕션에서 트리쉐이킹 하는 성질을 활용하여 process.env.NODE_ENV === "production"을 통해 빌드를 최적화했다.

사용자 정의 훅 테스트

```tsx
import { useState } from 'react'

import useEffectDebugger from '/useEffectDebugger'

function Test(props: { a: number; b: number }) {
  const { a, b } = props;
  useEffectDebugger('TestComponent', props)

  return (
    <>
        <div>{a}</div>
        <div>{b}</div>
    </>
  )
}

function App() {
  const [count, setCount] = useState(0)

  return (
    <>
        <button onClick={() => setCount((count) => count + 1)}>up</button>
        <Test a={count % 2 === 0 ? '짝수' : '홀수'} b={count} />
    </>
  )
}
```

- useEffectDebugger를 좌측 코드에 적용하면 다음과 같이 동작한다.

1. onClick을 통해 setCount가 동작한다.
2. props가 변경된다.
3. 콘솔에 다음과 같이 출력된다.

[useEffectDebugger] TestComponent

{ a: { before: '짝수', after: '홀수' },

b: { before: 0, after: 1 } }

사용자 정의 훅 테스트

```
import { renderHook } from '@testing-library/react'

import useEffectDebugger, { CONSOLE_PREFIX } from './useEffectDebugger'

const consoleSpy = jest.spyOn(console, 'log')
const componentName = 'TestComponent'

describe('useEffectDebugger', () => {
  afterAll(() => {
    // eslint-disable-next-line @typescript-eslint/ban-ts-comment
    // @ts-ignore
    process.env.NODE_ENV = 'development'
  })

  it('props가 없으면 호출되지 않는다.', () => {
    renderHook(() => useEffectDebugger(componentName))

    expect(consoleSpy).not.toHaveBeenCalled()
  })

  it('최초에는 호출되지 않는다.', () => {
    const props = { hello: 'world' }

    renderHook(() => useEffectDebugger(componentName, props))

    expect(consoleSpy).not.toHaveBeenCalled()
  })
```

- renderHook: 훅을 렌더링하기 위한 래퍼 객체
- 타입스크립트는 NODE_ENV를 Readonly로 설정하기 때문에 할당문을 강제로 작성했음
- renderHook으로 useEffectDebugger 호출

사용자 정의 훅 테스트

```
it('props가 변경되지 않으면 호출되지 않는다.', () => {
    const props = { hello: 'world' }

    const { rerender } = renderHook(() =>
      useEffectDebugger(componentName, props),
    )

    expect(consoleSpy).not.toHaveBeenCalled()

    rerender()

    expect(consoleSpy).not.toHaveBeenCalled()
  })
```

- rerender: 훅을 다시 호출하는 함수

사용자 정의 훅 테스트

```
it('props가 변경되면 다시 호출한다.', () => {
    const props = { hello: 'world' }

    const { rerender } = renderHook(
      ({ componentName, props }) => useEffectDebugger(componentName, props),
      {
        initialProps: {
          componentName,
          props,
        },
      },
    )

    const newProps = { hello: 'world2' }

    rerender({ componentName, props: newProps })

    expect(consoleSpy).toHaveBeenCalled()
  })
```

- initialProps: render하는 훅 함수의 초깃값

- 여기서 값을 변경해서 rerender하면 값을 변경해서 호출할 수 있다.

사용자 정의 훅 테스트

```
it('process.env.NODE_ENV가 production이면 호출되지 않는다', () => {
    // eslint-disable-next-line @typescript-eslint/ban-ts-comment
    // @ts-ignore
    process.env.NODE_ENV = 'production'

    const props = { hello: 'world' }

    const { rerender } = renderHook(
      ({ componentName, props }) => useEffectDebugger(componentName, props),
      {
        initialProps: {
          componentName,
          props,
        },
      },
    )

    const newProps = { hello: 'world2' }

    rerender({ componentName, props: newProps })

    expect(consoleSpy).not.toHaveBeenCalled()
  })
})
```

- production 단계에서 호출되는지 확인