[스파르타코딩클럽] 리액트 심화반 4주차

[수업 목표]

1. 에러 처리에 대해 배워본다.
2. 리액트에서 에러 처리를 다뤄본다.
3. 테스트 코드를 작성해본다.

[목차]

---

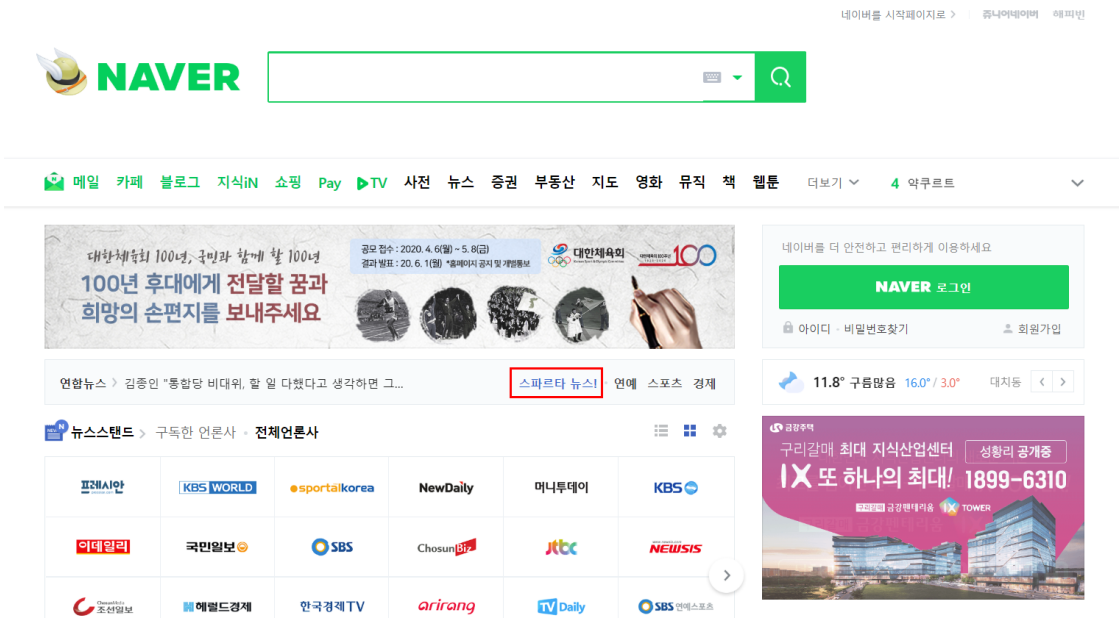
01. 오늘 배울 것

▼ 1) 에러란, 에러 처리, 테스트

▼ 1. 에러란

> 👉 쓰레드에 대해 들어보셨나요? 쓰레드는 쉽게 말해 일꾼이에요.
> 자바스크립트는 싱글 쓰레드 기반으로 동작하는 언어입니다.
> 다른 언어를 배워봤다면 싱글 쓰레드와 비동기를 함께 듣자마자 오잉? 하셨을 수 있어요.
> 일꾼이 하나인데 어떻게 비동기가 가능한 지 알아봅시다. 😄

> 👉 개념을 이해했다면 좀 더 단어를 정확히 해볼까요?
> 프로그램의 실행부터 종료까지 한 작업을 프로세스라고 해요.

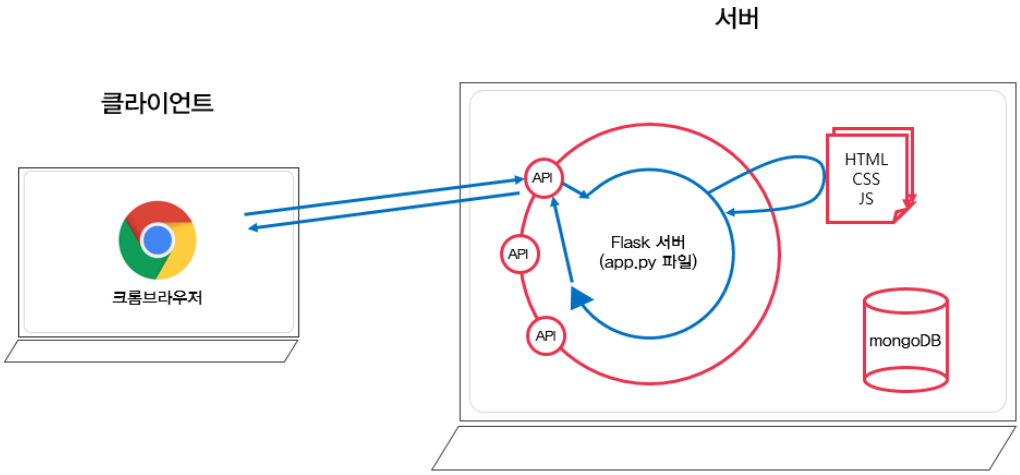


▼ 2. 에러 처리

👉 네! 우리가 보는 웹페이지는 모두 서버에서 미리 준비해두었던 것을 "받아서",
"그려주는" 것입니다. 즉, 브라우저가 하는 일은 1) 요청을 보내고, 2) 받은 HTML 파일을 그려주는 일 뿐이죠.

👉 근데, 1)은 어디에 요청을 보내냐구요? 좋은 질문입니다. 서버가 만들어 놓은 "API"라는 창구에 미리 정해진 약속대로 요청을 보내는 것이랍니다.

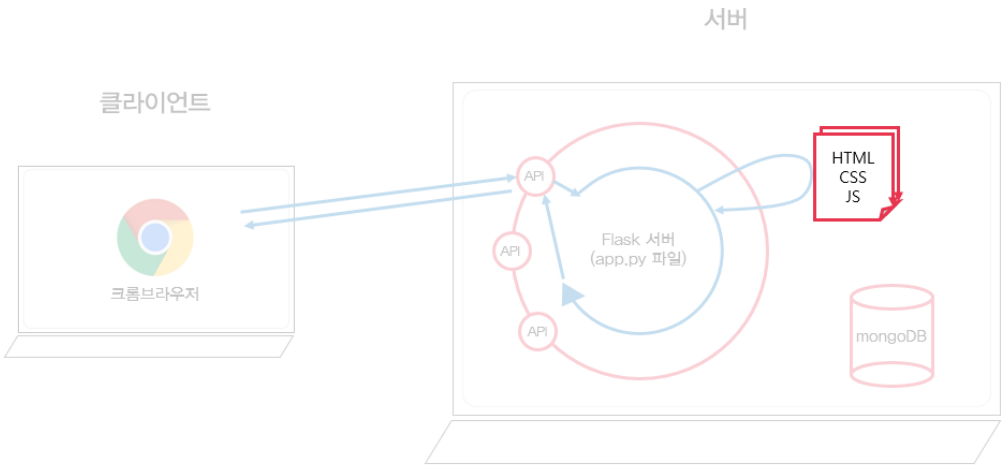
예) https://naver.com/
→ 이것은 "naver.com"이라는 이름의 서버에 있는, "/" 창구에 요청을 보낸 것!



▼ 3. 테스트

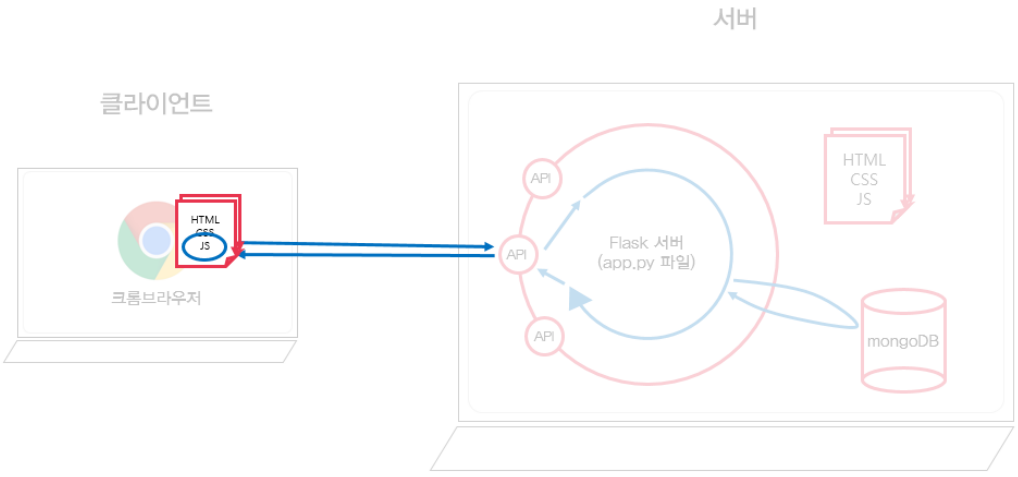
▼ 1주차: HTML, CSS, Javascript

> 👉 오늘은 HTML과 CSS를 배우는 날! 즉, 4주차에 내려줄 HTML파일을 미리 만들어 두는 과정입니다. + 또, 2주차에 자바스크립트를 능숙하게 다루기 위해서, 오늘 문법을 먼저 조금 배워둘게요!



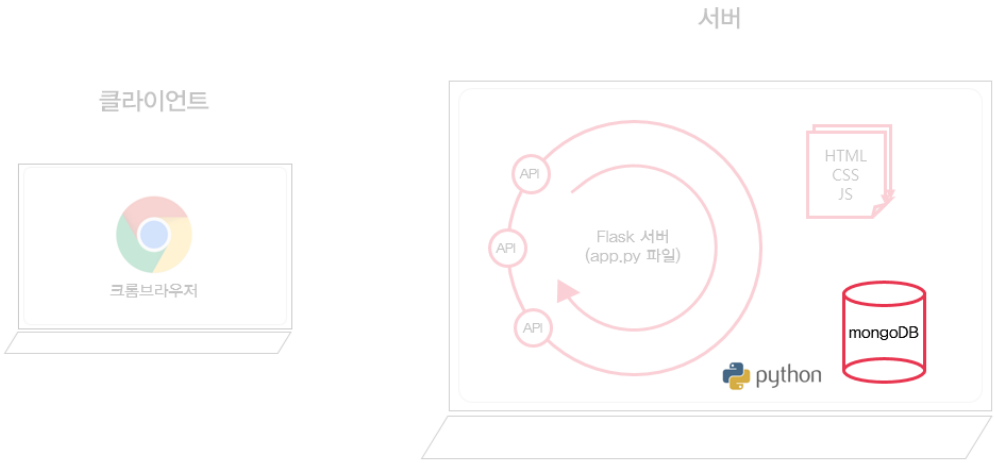
▼ 2주차: jQuery, Ajax, API

> 👉 오늘은 HTML파일을 받았다고 가정하고, Javascript로 서버에 데이터를 요청하고 받는 방법을 배워볼거예요



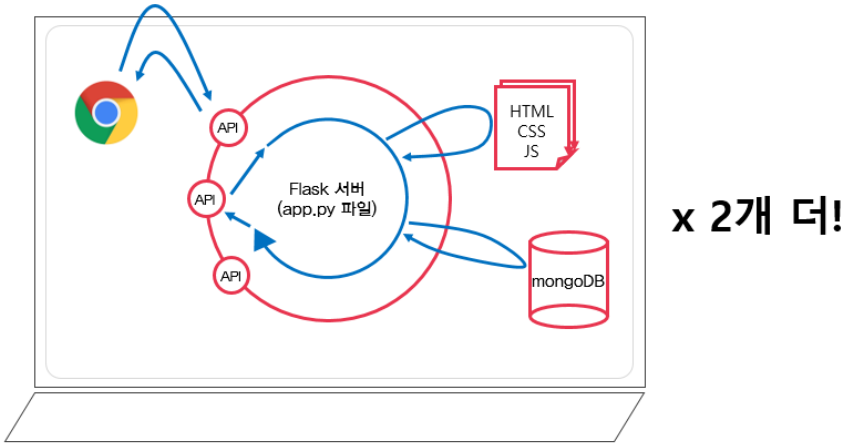
▼ 3주차: Python, 크롤링, mongoDB

👉 오늘은 드디어 '파이썬'을 배울거예요. 먼저 문법을 연습하고, 라이브러리를 활용하여 네이버 영화목록을 쫙 가져와보겠습니다. (기대되죠!)
+
그리고, 우리의 인생 첫 데이터베이스. mongoDB를 다뤄볼게요!



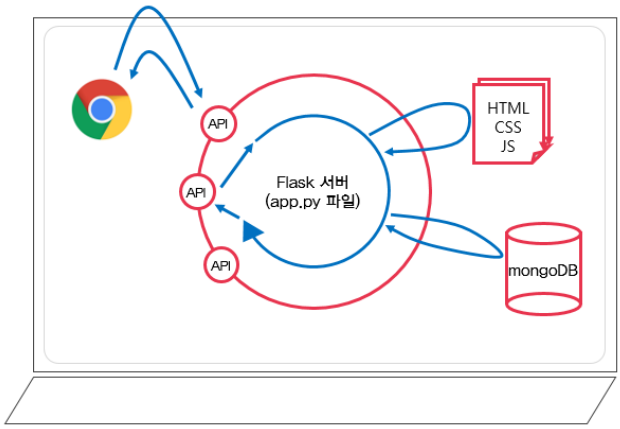
▼ 4주차: 미니프로젝트1, 미니프로젝트2

👉 오늘은 서버를 만들어봅니다! HTML과 mongoDB까지 연동하여,
미니프로젝트1, 2를 완성해보죠! 굉장히 재미있을 거예요!



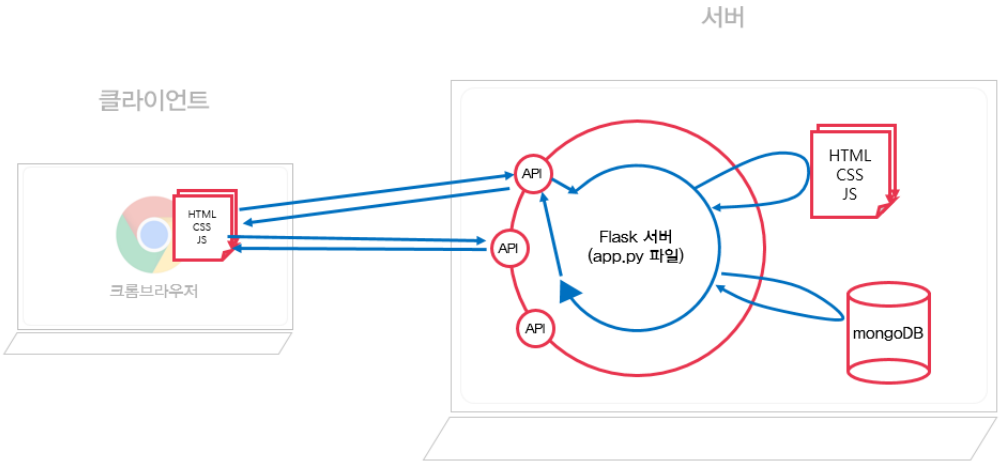
👉 나중에 또 이야기하겠지만 헷갈리면 안되는 것!
우리는 컴퓨터가 한 대 잖아요... 그래서 같은 컴퓨터에다 서버도 만들고, 요청도 할 거예요. 즉, 클라이언트 = 서버가 되는 것이죠.

이것을 바로 "로컬 개발환경"이라고 한답니다! 그림으로 보면, 대략 이렇습니다.



▼ 5주차: 미니프로젝트3, AWS

> 👉 오늘은 아직 익숙해지지 않았을 당신을 위해! 같은 난이도의 유사한 한 개의 프로젝트를 더 진행합니다.
>
> 그치만 우리 컴퓨터를 24시간 돌려둘수는 없잖아요!
>
> 그래서 두구두구.. 인생 첫 배포!를 해볼 예정입니다! 클라우드 환경에서 컴퓨터를 사고, 거기에 파일을 올려 실행해보겠습니다.



02. Try - Catch - Finally

▼ 2) 에러 하나 내보기

> 👉 **배포된 앱은 에러가 나면 어떻게 될까?**
> 콘솔의 빨간 메시지! 이제 제법 친해졌나요? 😊
> 개발 모드에서는 에러가 나면 화면에도 에러가 표시되지만 빌드 결과물은 그렇지 않습니다.
> 빈 화면만 쭉 표시되어 있어요.

- 아래 코드를 App.js에 추가한 후, 에러를 확인해봅시다.
  개발 환경과 빌드 파일 둘 다 확인해보세요!

```
const a = 1;
a = 222;
```

- 에러가 나도 스크립트가 죽지 않게 하려면?

> 👉 에러가 나면 빈 화면만 뜨는 이유는 스크립트가 더 이상 동작하지 않기 때문이에요.
> 스크립트가 죽지 않게 한다면 에러가 나더라도 어떤 화면을 계속 보여줄 수 있습니다!
> (에러가 난 부분은 보여줄 수 없더라도요!)

▼ 3) try-catch로 에러 던지고 잡기

- try - catch

> 👉 try - catch 문은 어떤 코드를 동작시켜보고(try), 에러가 나면 그 에러를 잡아(catch),
> 스크립트가 멈추지 않게 해주는 문법입니다. 😊
> 아래처럼 쓸 수 있어요!

```
try {
  const a = 1;
  a = 222;
} catch (err) {
  console.log(err);
}
```

- 동작 원리
  - try-catch에 진입하면 안의 코드를 실행합니다.
  - 에러가 없을 경우 catch 블럭을 무시하고, (건너뛰어요!)
  - 에러가 있을 경우, catch 블럭으로 제어 흐름이 넘어가요. (catch 블럭 코드 실행됩니다!)

▼ 4) finally 구문

> 👉 try - catch는 finally라는 블록을 하나 더 추가해서 사용할 수 있습니다!
> finally는 try에 적은 코드 구문이 오류가 나도, 나지 않아도 항상 실행됩니다.
> try와 catch에 공통적으로 들어가야만 하는, 오류가 나건 나지 않건 꼭 해야하는 작업이 있다면 finally에서 하면 됩니다!

- finally 써보기

```
try {
  if( window.confirm("에러가 났나요??")){
    웅웅();
  }
} catch (err) {
  console.log(err);
} finally {
  console.log("난 항상 실행될거야!");
}
```

⚠️ try - catch가 모든 에러를 잡아줄까?
답은 아니오입니다.
에러에는 여러 종류가 있어요. {}나 ()가 덜 닫혔을 때 나는 에러, 방금의 예시처럼 상수 값을 바꿔서 나는 에러 등등등!
try - catch는 오직 읽을 수 있는 코드에 한하여 에러를 잡을 수 있습니다.
자바스크립트는 코드를 실행하기 전에 미리 한 번 읽어보고 이게 다 말이 되는 건가~ 내가 실행할 수 있는 건가~하고 확인을 거치는데요, 이 과정에서 오류가 나면 parse-time에러가 납니다.
이런 parse-time 에러는 try - catch로 잡을 수 없습니다. 😂
try - catch문은 오직 오직 런타임 에러에만 동작해요.

03. Error 다루기

▼ 5) 자바스크립트의 Error 객체 살펴보기

👉 **Error 객체**
자바스크립트에서는 Error도 객체입니다.
이미 우리가 여러 차례 만났던 에러를 떠올려볼까요? Error, SyntaxError, ReferenceError, TypeError가 가장 자주 만난 에러였을 겁니다.
이런 에러들을 표준 에러 객체라고 해요.
표준 에러 객체들은 전부 생성자를 사용해 직접 만들고 커스텀할 수 있습니다!

- 예시 :

```
let my_new_error = new Error("에러 메시지야!");
let my_new_syntaxError = new SyntaxError("신텍스 에러 메시지야!");
```

- Error의 프로퍼티
  - 에러 객체는 기본적으로 name과 message 두 개의 프로퍼티를 가지고 있어요.
  - 에러를 생성할 때 별도의 프로퍼티 지정을 하지 않는다면, name과 message는 같은 값으로 들어갑니다. (name이 생성자와 동일한 값을 가지게 됩니다! 위 경우, Error에서는 name이 Error가 될 거예요.)

▼ 6) 나만의 Error 객체 만들기

👉 **에러를 만들고 던져보자!**
임의의 에러를 일으키는 걸 throw라고 해요.
`throw new Error("error 났어!");` 처럼 사용할 수 있습니다.

- 에러 만들고 던져보기

```
const data = {my_cat: "펄이", ageeee: "20"};

try {
  let perl_age = data.age;

  if(!perl_age){
      throw new Error("고영 나이를 가져오질 못했어요!");
  }
} catch(err) {
  console.log(err);
}
```

👉 Error 객체는 숫자, 문자열 등 뭐든 넣을 수 있어요.
하지만 살펴보았던 표준 에러와 호환을 위해 name, message 프로퍼티를 꼭꼭 넣어주는 게 좋습니다.

🔥 throw는 try에서만 할 수 있을까?
catch에서도 할 수 있습니다! 커스텀 Error를 다룰 때에는 try에서 코드를 적으며 예상한 에러를 잡아서 우리가 원하는 방식으로 처리하라고 지정할 수 있게 됩니다. 그러기 위한 데이터도 에러를 통해 넘길 수 있고요.

그런데 언제나 예상치 못한 에러가 날 수 있잖아요!
그럴 땐 catch에서 throw를 사용해서 error를 try - catch문 바깥으로 던져버리는거예요.
그럼 바깥 쪽의 try - catch문에서 예상치 못했던 그 에러를 잡아 처리할 수 있습니다.
(물론 이렇게 하기 위해선 try - catch 안에 try - catch가 또 자리잡아야겠죠! 😊)

04. ErrorBoundary

▼ 7) 리액트에서 에러가 나면 어떻게 될까?

👉 자바스크립트는 에러가 나면 코드 실행을 멈춥니다.
리액트에서 자바스크립트 코드 실행이 멈추게 되면?
새하얀 화면만 하염없이 보게 돼요. 🥲

- 자바스크립트 에러가 전체 어플리케이션을 중단시키게 되는 것!
- 리액트는 이런 문제를 해결하려고 16버전부터 에러 바운더리라는 것을 도입했습니다.

▼ 8) ErrorBoundary

👉 Error Boundary
에러 경계라고도 해요.
에러 경계는 어떤 컴포넌트에서 자바스크립트 에러가 났을 경우, 멈춰버린 컴포넌트를 대신해서 fallback UI를 보여주도록 하는 리액트 컴포넌트입니다.

👉 **error boundary, 어떻게 쓸 수 있나?**
기본적으로 error boundary는 생명주기 함수인 getDerivedStateFromError()와 componentDidCatch()를 정의한 클래스형 컴포넌트를 만들어야 합니다. (둘 중 하나를 정의해서 만든 클래스형 컴포넌트가 에러 경계가 되어줘요.)

getDerivedStateFromError()를 사용해서 풀백 UI를 보여주고, componentDidCatch()를 사용해서 에러 내용을 기록하는 형태로 사용합니다.

하지만 react-error-boundary라는 패키지를 사용하면 굳이 직접 만들지 않아도 잘만들어진 에러경계 컴포넌트를 가져다 쓸 수 있어요. (우린 이렇게 할거예요!)

- 특정 컴포넌트를 error boundary로 감싸두면 그 아래의 컴포넌트 트리에서 에러가 발생할 경우 그 에러를 잡아냅니다.
- 다만 못잡는 에러가 몇 가지 있어요!
  - 이벤트 핸들러에서 오류가 날 경우
  - 비동기 코드에서 에러가 날 경우
  - 서버 사이트 렌더링을 하고 있을 경우
  - 자식 컴포넌트가 아니라 에러 경계 자체에서 에러가 날 경우
- 사용방법
  - react-error-boundary 패키지에서 ErrorBoundary를 가져와서 이런 식으로 사용하면 됩니다.

```
import {ErrorBoundary} from "react-error-boundary";

const ErrorFallback = () => {
  return (<div>에러났어!</div>)
}

<ErrorBoundary FallbackComponent={ErrorFallback}>
  <자식컴포넌트 />
</ErrorBoundary>
```

  - try - catch의 catch와 유사하게 동작하는 컴포넌트라고 생각하면 됩니다.
  - 에러 경계로 감싼 것 바깥의 에러는 잡지 못해요!
- 해보자!

👉 Cat 컴포넌트를 만들고 에러 경계로 감싸줍시다.
그리고 Cat에서 에러를 내보는거예요. 😊
에러 경계가 오류가 났을 때 어떻게 대처하는 지 눈으로 확인해보면 좀 더 감이 잘 올거예요.

  - react-error-boundary를 설치합시다.

```
yarn add react-error-boundary
```

  - ErrorBoundary를 적용하고,

```
// App.js
import React from "react";
import { ErrorBoundary } from "react-error-boundary";
import './App.css';
import Cat from "./Cat";


const ErrorFallback = () => {
  return (<div>에러났어!</div>)
}

function App() {
  return (
    <div className="App">
      <ErrorBoundary FallbackComponent={ErrorFallback}>
        <Cat/>
      </ErrorBoundary>
    </div>
  );
}

export default App;
```

- Cat 컴포넌트에서 에러를 내봅시다.

```
// Cat.js
const Cat = () => {
    const cat_name = "펄이";
    return (
        <div>{cat_name.cat.cat}</div>
    )
}

export default Cat;
```

▼ 9) ErrorBoundary에 커스텀 에러 적용하기

> 👉 이번엔 커스텀 에러를 한 번 적용해봐요!

- Cat 컴포넌트에서 커스텀 에러를 던지게 하고,

```
const Cat = () => {
    const cat_name = "멍멍이";

    if(cat_name !== "펄이"){
        throw new Error("펄이가 아니예요!");
    }
    return (
        <div>{cat_name}</div>
    )
}

export default Cat;
```

- FallbackComponent에서 에러 메시지를 로깅하게 만들어줍시다.

```
import React from "react";
import { ErrorBoundary } from "react-error-boundary";
import "./App.css";
import Cat from "./Cat";
```

```
const ErrorFallback = (err) => {
  console.log(err);
  return <div>에러났어!</div>;
};

function App() {
  return (
    <div className="App">
      <ErrorBoundary FallbackComponent={ErrorFallback}>
        <Cat />
      </ErrorBoundary>
    </div>
  );
}

export default App;
```

👉 이런 식으로 커스텀 에러와 에러 바운더리를 적절히 쓰면
실제로 에러가 아닌 우리 로직 상의 에러를 처리할 때 편리합니다. 😊

05. Axios interceptors 써보기

▼ 10) axios interceptor란?

👉 axios의 interceptor는 .then()이나 .catch()로 처리되기 전 요청이나 응답을 가로챌 수 있습니다!
요청을 보낼 때 토큰을 붙여나가고 싶거나, 응답을 받아서 처리하기 전(.then()이나 .catch()가 실행되기 전) 가로채서 오류 처리를 먼저하고 싶을 때 등 다양하게 활용할 수 있어요!

- 인터셉터 써보기
  - axios 객체 만들고,

```
// shared/axios.js
import axios from "axios";

const instance = axios.create({
  headers: {"Content-Type": "application/json"},
  timeout: 5000,
});

export default instance;
```

  - 인터셉터를 써보자!

👉 요청 보내기 전에 가로채기를 해볼거예요!
`악시오스_객체.interceptors.request.use();` 로 사용할 수 있습니다 😊

```
// shared/axios.js
...
```

```
instance.interceptors.request.use(config => {
  const _conf = {headers: {"Content-Type": "application/json", "X-Auth-Token":"1234"}, timeout: 3000};

  console.log(config, _conf);
  return _conf;
}, err => {
  return Promise.reject(err);
});
...
```

- 요청을 하나 보내기

> 👉 전에 만들었던 mock api(http://localhost:5001/sleep_times)로 요청을 보내볼거예요!

```
// App.js
...
instance.get("http://localhost:5001/sleep_times");
...
```

- 개발자 도구 → 네트워크 탭에서 확인해보자!

> 👉 요청이 보내지기 전에 헤더를 가로채서 바꿔줬어요!
> 우리가 로그인 한 후 api 요청마다 토큰을 보내줄 때에도 인터셉터를 사용해서 이렇게 보내줄 수 있습니다. 😊

▼ 11) json-server로 500, 404 등 에러 내려주기

> 👉 이번엔 response에서 에러를 가로채오는 것도 해볼까요?
> 그 전에 response에서 에러를 받아올 수 있도록, mock api에서 에러를 주게끔 만들어봅시다!

- mock api 좀 더 잘 써보기 (독스를 참고해요! 독스보러가기 →)

```
//server.js
const jsonServer = require("json-server");
const server = jsonServer.create();
const router = jsonServer.router("db.json");
const middlewares = jsonServer.defaults();

server.use(middlewares);

server.use((req, res, next) => {
  next();
});

server.use(router);
server.listen(5001, () => {
  console.log("JSON Server is running");
});
```

- 500에러 내려주기

```
const jsonServer = require("json-server");
const server = jsonServer.create();
const router = jsonServer.router("db.json");
```

```
const middlewares = jsonServer.defaults();

server.use(middlewares);

server.get("/geterror", (req, res) => {
    res.status(500).jsonp({error: "500에러야!"});
})
server.use((req, res, next) => {
  next();
});

server.use(router);
server.listen(5001, () => {
  console.log("JSON Server is running");
});
```

- mock api 실행하기

> 👉 js 파일을 노드 런타임에서 실행해줄 때는 node [실행할 파일 명]으로 실행할 수 있어요.
> 터미널에서 아래 명령어를 입력해봅시다!
> **(실행하기 전에, 기존 5001 포트를 쓰고 있던 건 종료해줘야겠죠!)**

```
node server.js
```

▼ 12) interceptor로 api 에러 가로채기

> 💡 이제 response를 인터셉트 하고 잘 가로챘는지 확인해봐요. 😊

- response interceptors 적용하기

```
...
instance.interceptors.response.use(
  config => {
    return config;
  },
  err => {
    console.log(err);
    return Promise.reject(err);
  },
);
...
```

- 요청을 하나 보내기

> 👉 전에 만들었던 mock api(http://localhost:5001/geterror)로 요청을 보내볼거예요!

```
// App.js
...
instance.get("http://localhost:5001/geterror");
...
```

- 개발자 도구 → 네트워크 탭에서 확인해보자!

👉 응답을 처리하기 전에 가로채봤습니다!
여기에서 500일 때는 무조건 이런 alert을 보여줘! 같은 식으로 구성해주면 매 통신 오류마다 직접 처리하는 일이 줄어들어 편해요!

06. 테스트 1 - 테스트란?

▼ 13) 테스트란?

👉 **테스트란?**
테스트는 우리가 만든 코드가 잘 돌아가는 지 확인하는 과정입니다.
지금껏 우리가 콘솔로 로그 남기면서 값이 잘 왔나 확인했던 것이나, 브라우저에서 잘 뜨나 확인했던 것 모두 테스트예요.

👉 **프로젝트 규모가 커진다면?**
지금껏 만든 조그만 예제들처럼 확인할 게 적다면 배포하기 전에 모두 확인할 수 있어요. 시간도 그리 오래 걸리지 않을거고요.
하지만 프로젝트 규모가 커지면 모두 직접 확인하기는 무리가 있습니다.
그래서 코드로써 내가 만든 코드가 확실해!라는 테스트를 거치는 게 좋습니다. → 이런 걸 두고 테스트 자동화라고 합니다.

- 테스트 종류
  - 단위 테스트 : 유닛 테스트라고도 합니다. 어떤 코드 한 블럭이 정상 동작하는 지 보기 위한 테스트입니다.
    리액트에서는 어떤 컴포넌트가 제대로 렌더링 되는 지 확인하는 것도 단위 테스트로 볼 수 있습니다.
    최대한 간결해야 합니다!
  - 통합 테스트 : 단위 테스트의 묶음입니다.
    예를 들어 로그인 통합 테스트를 한다면, ID, PW를 가져오는 함수와 API 요청을 보내는 함수, 응답을 받아 페이지를 이동시키는 함수를 묶어서 테스트하는 거예요.
  - 종단간 테스트 : E2E 테스트라고도 합니다. 전체 앱의 흐름이 매끄럽게 이어지는 지 테스트합니다.
- 리액트에서의 테스트
  - 컴포넌트 트리

👉 테스트 라이브러리를 고를 땐?
- 반복 작업을 최대한 줄이고 싶은 지
- 실제 환경과 최대한 유사하게 짜볼 건지
- 어디까지 테스트해볼 건지

- 우리가 써볼 테스트 툴은?
  → Jest와 react-testing-library!

> 👉 CRA로 프로젝트를 만들었다면 Jest와 React Testing Library는 이미 프로젝트에 포함되어 있습니다. 😊
> **(Jest 공식문서 보러가기→)**
> **(React Testing Library 공식문서 보러가기→)**

07. 테스트 2 - Jest 1

▼ 14) Jest를 써보자!

> 👉 Jest는 자바스크립트 테스트를 위해 굉장히 널리 쓰이는 테스팅 라이브러리예요.
> 설치도 쉽고 사용법도 굉장히 간단합니다. jest로 할 수 있는 게 정말 많은데도요!
> 아참, 할 수 있는 게 많은만큼 봐야할 기능도 많은 편이니 꼭 한 번은 독스를 읽어보세요!
> **(Jest 공식문서 보러가기→)**

▼ 프로젝트 만들기

```
yarn create react-app test_ex
```

▼ 테스트 코드 써보기

> 👉 jest로 간단한 테스트 코드를 써보면서 기본 사용법을 알아볼게요.

- 더하기 함수 만들기

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

export default add;
```

- 더하기 함수를 테스트하는 테스트 케이스 만들기

```
import add from "./calculator";

it("add test", () => {
    expect(add(1,2)).toBe(4);
});
```

- yarn test로 테스트 하기

👉 yarn test를 입력해서 테스트를 해봅시다.
jest가 ~~.test.js 파일에 넣은 테스트를 실행해줄거예요.
toBe()의 값을 바꿔가면서 테스트 성공과 실패시 각각 어떻게 나오는 지 확인해봐요!

```
yarn test
```

▼ 용어 정리

- test suite : 테스트 케이스를 묶어서 테스트 스위트라고 합니다.
- test runner : 테스트 러너는 테스트 스위트를 실행할 수 있게 해주는 역할을 합니다.
- it() : 테스트 케이스를 만드는 함수
- describe() : 같은 맥락 상의 테스트를 그룹화 하는 함수
  저는 어떤 한 기능을 묶을 때 많이 씁니다.
- expect().toBe() : expect()의 인자로는 테스트할 내용을, toBe()의 인자로는 기대값(잘 돌아갔다면 나와야하는 값)을 넘겨줍니다.

  expect()에 넘겨준 내용이 toBe()에 넘겨준 조건과 일치하는 지를 확인해서 테스트 하는거예요.

08. 테스트 3 - Jest 2

▼ 15) 여러 테스트 케이스를 묶어서 테스트 해보자!

👉 간단한 테스트를 해봤으니, 이번에는 조금 더 복잡한 테스트를 해봅시다!
mock api에서 숫자가 든 배열을 가져온 다음, 배열 안의 값을 더해주는 함수를 만들고 테스트 해봐요.
테스트 케이스를 묶어 테스트 슈트를 만들고 테스트 슈트를 테스트해봅시다.

- mock API에 numbers를 추가해요!

```
//db.json
{
...
"numbers": [1,2,3,4,5],
...
}
```

- mock API를 사용해서 데이터를 가져오는 함수를 하나 만들고,

```
export const getNumbers = async () => {
  return fetch("http://localhost:5001/numbers");
};
```

- mock API와 잘 통신하고 있는 지 테스트 해봐요. 😊

👉 getNumbers()는 프라미스를 반환하죠?
Promise를 리턴하게 되면 test는 이 프라미스가 resolve되었나 rejesct 되었나로 테스트 성공 실패 여부를 가릅니다.

```
test("get response!", () => {
    return getNumbers();
  });
```

- 데이터도 잘 가져왔는 지 테스트 해봅시다.

👉 테스트 내에서도 async - await를 사용할 수 있어요. 😊

```
describe("add array number test", () => {
  test("get response!", () => {
    return getNumbers();
  });

  test("get numbers!", async () => {
    let res = await getNumbers();
    let data = await res.json();

    console.log(data);

    expect(data).toStrictEqual([1, 2, 3, 4, 5]);
  });
});
```

- 배열 안의 숫자를 모두 더하는 함수를 만들고,

```
export const addArrayNumbers = (arr) => {
  let sum = 0;

  arr.map((a) => {
    sum = sum + a;
  });

  return sum;
};
```

- mock API에서 가져온 데이터를 모두 더해서 확인하는 테스트 코드를 짜봅시다!

```
import { add, getNumbers, addArrayNumbers } from "./calculator";

it("add test", () => {
  expect(add(1, 2)).toBe(3);
});

describe("add array number test", () => {
  test("get response!", () => {
    return getNumbers();
  });

  test("get numbers!", async () => {
    let res = await getNumbers();
    let data = await res.json();

    console.log(data);
```

```
    expect(data).toStrictEqual([1, 2, 3, 4, 5]);
  });

  test("add numbers!", async () => {
    let res = await getNumbers();
    let data = await res.json();

    expect(addArrayNumbers(data)).toBe(15);
  });
});
```

- 이제 테스트 해봐요!

```
yarn test
```

09. 테스트 4 - RTL 1

▼ 16) React-testing-library란?

> 👉 React testing library는 컴포넌트 렌더링을 테스트하기 위한 테스팅 툴입니다.
> RTL은 멋진 철학을 가진 테스팅 도구예요.
> 우리가 만드는 건 결국 사용자가 보기 위한 거니까 **사용자 경험과 유사한 방식의 테스트를 하자**!가 모토 거든요!
> **(React Testing Library 공식문서 보러가기→)**

- `<div>안녕 여러분!</div>`이 나오도록 테스트한다면, RTL은 `안녕 여러분!`이 화면에 보이는 게 맞니?에 집중합니다.
- 리액트에서는 RTL과 Jest를 같이 써서 테스트합니다. (RTL이 Jest를 포함하고 있어요.)
- 사용법은 Jest 사용법과 거의 같습니다.

▼ 17) 단순 스냅샷 확인하기

> 👉 **스냅샷이란?**
> 화면 캡쳐를 생각하세요! 캡쳐하면 어떤 순간의 내 화면이 그대로 남잖아요? 그것처럼 스냅샷은 특정 순간을 코드로 남겨두는 거예요.
> 오늘 우리가 말하는 스냅샷은 렌더링할 컴포넌트의 화면이 될겁니다. 😊
>
> **스냅샷 테스트는,**
> 성공했을 경우의 화면하고 테스트하는 시점의 화면을 비교하는 테스트예요.

> 👉 오늘 테스트를 위한 프로젝트 초안은 제가 미리 만들어두었습니다!
> 아래 코드스니펫에서 가져다가 사용하세요!
> - zip 파일을 다운로드 하신 후, 원하는 경로에 프로젝트를 넣어주세요.
> - vscode로 프로젝트를 여신 후, 터미널에서 `yarn install` 명령어로 필요한 패키지를 모두 설치해주세요.

▼ [코드 스니펫] 테스트용 프로젝트.zip

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/4119e56e-2267-499c-bc6d-0d0d812359d3/test_ex_프로젝트.zip

- 코드 살펴보기

> 👉 고양이, 강아지 페이지가 있는 프로젝트네요!
> 강아지 페이지에서는 고양이 페이지로 이동하는 링크가 있고,
> 고양이 페이지에서는 고양이 마리 수를 늘리는 버튼이 하나 있습니다.

- 테스트할 게 뭐가 있을 지 생각 먼저!

> 👉 고객님 입장에서 생각해봐요. 이 프로젝트에서 고객이 어떻게 움직일까?를 생각하면서 뭘 테스트해야하나 생각해야해요. 저희는 아래를 테스트합니다.
> - 페이지가 잘 나오고 있을까?
> - 페이지 이동이 잘 되나? (= 링크가 잘 걸렸을까?)
> - 버튼을 눌렀을 때 숫자가 잘 올라갈까?

- 페이지가 잘 나오는 지 테스트하기

> 👉 이제야 왜 제가 늘 index.js에 BrowserRouter를 붙이는 지 말씀드릴 수 있게 되었네요.
> 라우팅을 포함하는 프로젝트의 테스트를 진행할 때에는 우리가 늘 쓰는 브라우저라우터가 아니라 메모리 라우터라는 걸 사용해서 라우팅처리를 해줘야해요!
> 이때 App.js에서 BrowserRouter를 붙이면 테스트 코드를 위해 MemoryRouter 붙이기가 곤란해지니까 늘 분리해서 적어왔던거요. 😊

  - 테스트 코드부터 짜고

```
// Dog.test.js
import { render } from "@testing-library/react";
import App from "./App";
import { MemoryRouter } from "react-router-dom";

describe("Dog", () => {
  test("dog page 잘 뜨나?", () => {
    render(
      <MemoryRouter>
        <App />
      </MemoryRouter>
    ).debug(); // .debug()를 붙이면 터미널에서 스냅샷을 볼 수 있어요!
  });
});
```

  - yarn test로 테스트를 실행해줍시다!

```
yarn test
```

10. 테스트 4 - RTL 2

▼ 18) 클릭 이벤트 테스트하기

> 👉 이번엔 클릭 이벤트를 테스트해보죠!
> 테스트 진행을 끊어서 살펴보면, 컴포넌트를 띄운다 → 액션을 발생 시킨다 → 결과를 본다! 이 순서로 진행됩니다.

- 클릭할 수 있는 요소 찾기
  - 쿼리를 사용해서 요소를 찾아볼거예요. 여기서 쿼리는 렌더링 된 DOM 노드에 접근해서 요소를 가져오는 메서드를 말해요. 대표적으로 getAllByRole, getAllByText 등이 있어요. 자세한 쿼리는 공식 문서에서 확인해보세요!

```
// Cat.test.js
import { render, screen } from "@testing-library/react";
import { Cat } from "./Cat";
import { MemoryRouter } from "react-router-dom";

describe("Cat", () => {
  test("cat page 잘 뜨나?", () => {
    render(
      <MemoryRouter>
        <Cat />
      </MemoryRouter>
    );

    const button = screen.getByRole("button", { name: "고양이 추가하기" });

    console.log(button);
  });
});
```

- 클릭해봅시다!

```
import userEvent from "@testing-library/user-event";
...
const button = screen.getByRole("button", { name: "고양이 추가하기" });
userEvent.click(button);
...
```

- 클릭되면 숫자가 하나 올라가야 하죠! 테스트해봅시다.

```
import { render, screen } from "@testing-library/react";
import { Cat } from "./Cat";
import { MemoryRouter } from "react-router-dom";

import userEvent from "@testing-library/user-event";

describe("Cat", () => {
  test("cat page 잘 뜨나?", () => {
    render(
      <MemoryRouter>
        <Cat />
      </MemoryRouter>
    );

    const button = screen.getByRole("button", { name: "고양이 추가하기" });

    userEvent.click(button);
    // 이 주석을 풀어보세요 :) 그럼 현재 화면에서의 스냅샷을 볼 수 있어요.
```

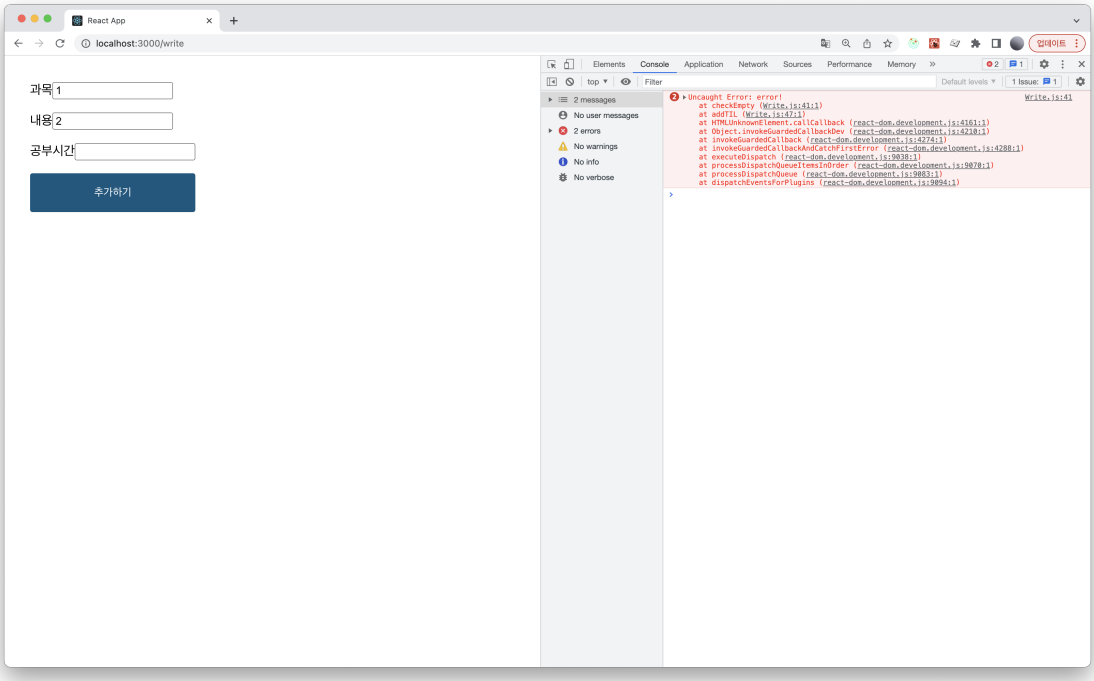
```
    // screen.debug();

    // 주석을 풀고 보셔요! 테스트 실패겠지요!
    // expect(screen.getAllByText("고양이가 2마리 있어요!")).toBeTruthy();
    expect(screen.getAllByText("고양이가 2 마리 있어요!")).toBeTruthy();


  });
});
```

11. 끝 & 숙제 설명

이번 주에는 내가 만든 코드가 정말 잘 동작하는 지 확신하는데 꼭 필요한 내용들을 배웠어요. 단순히 코드가 잘 돌아간다에서 끝내지 않게요! 😊

[숙제]
이번 주에 배운 내용을 바탕으로 우리 TIL 페이지에 에러처리를 해봅시다!
아래 화면처럼, 꼭 입력해야하는 내용 중 무언가가 입력되지 않았다면 커스텀 에러가 발생하게 해보고 에러가 나면 alert으로 빠진 내용이 있다고 알려줍시다.

▼ 예시 화면



https://s3-us-west-2.amazonaws.com/secure.notion-static.com/12fee18b-9c59-4020-a9ff-a42a5662e8bb/homework_w4.zip