



# [스파르타코딩클럽] 리액트 심화반 2주차

## [수업 목표]

1. 자바스크립트의 동작방식과 문법을 배운다.
2. 리액트 훅에 대해 배우고 적재적소에 쓸 수 있다.
3. 상태관리에 대해 배우고, 상태관리 라이브러리를 선택해 쓸 수 있다.
4. 라우팅 처리를 할 수 있다.

## [목차]

- 01. 오늘 배울 것
- 02. Hook - (1) Hook이란?
- 03. Hook - (2) 자주 써먹는 Hook
- 04. Quiz\_ 텍스트 입력기를 만들어보자!
- 05. Custom Hook
- 06. 자바스크립트와 동시성
- 07. Promise
- 08. async, await
- 09. 상태관리
- 10. 상태관리 해보기 - (1) ContextAPI()
- 11. 상태관리 해보기 - (2) Redux 1
- 12. 상태관리 해보기 - (3) Redux 2
- 13. 라우팅
- 14. 끝 & 숙제 설명

## 01. 오늘 배울 것

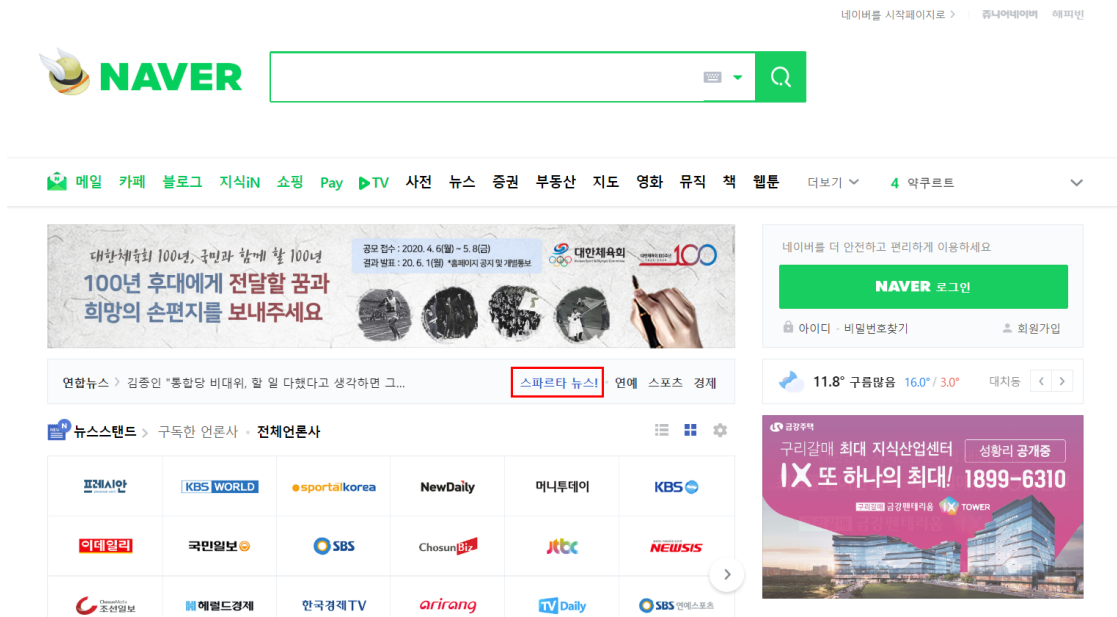
### ▼ 1) 훅, 커스텀 훅, 프라미스, 상태관리

#### ▼ 1. 동기와 비동기



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 알아보시다. 😊

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



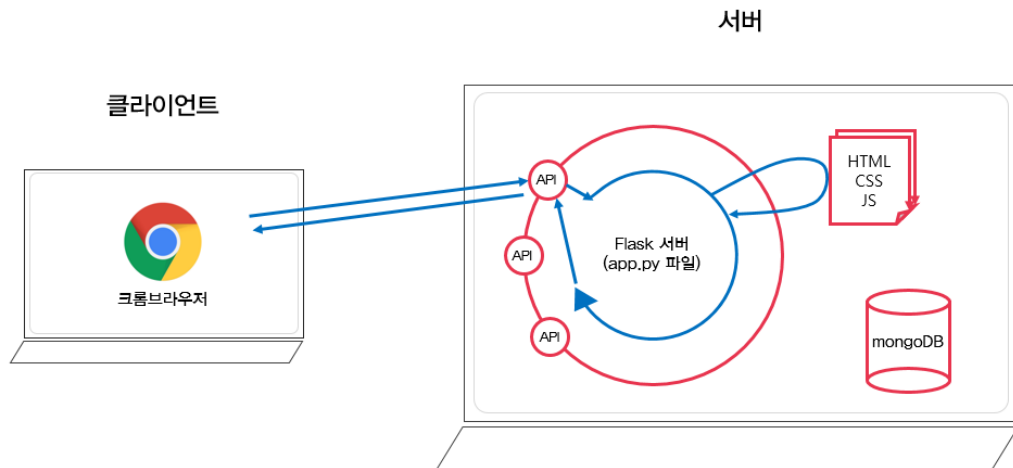
## ▼ 2. 상태관리

👉 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!



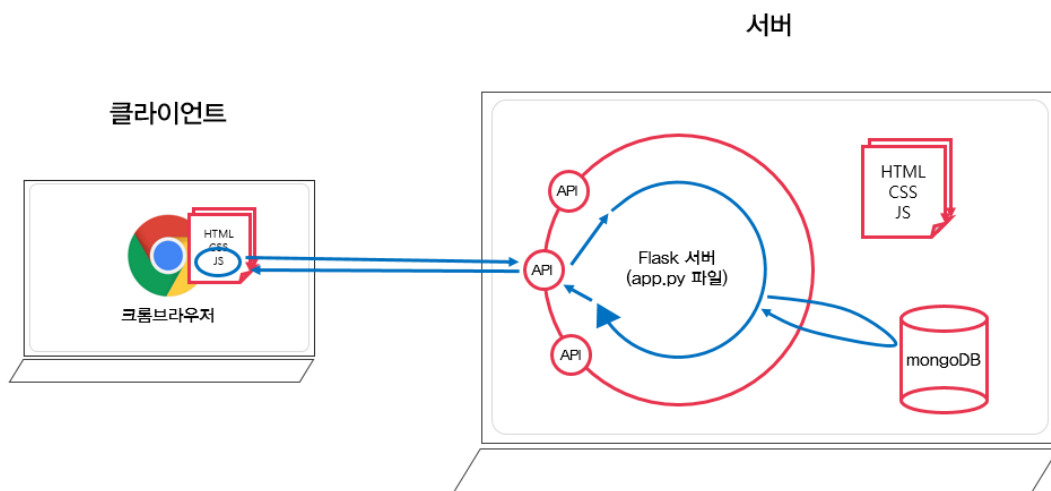
### ▼ 3. 네트워크 요청

☞ 앳, 그럼 항상 이렇게 HTML만 내려주냐구요?  
아뇨! 데이터만 내려 줄 때가 더~ 많아요.

사실 HTML도 줄글로 쓰면 이게 다 '데이터'아닌가요?

☞ 자, 공연 티켓을 예매하고 있는 상황을 상상해봅시다!  
좌석이 차고 꺼질때마다 보던 페이지가 리프레시 되면 난감하겠죠??

이럴 때! 데이터만 받아서 받아 끼우게 된답니다.



☞ 데이터만 내려올 경우는, 이렇게 생겼어요!  
(소곤소곤) 이런 생김새를 JSON 형식이라고 합니다.

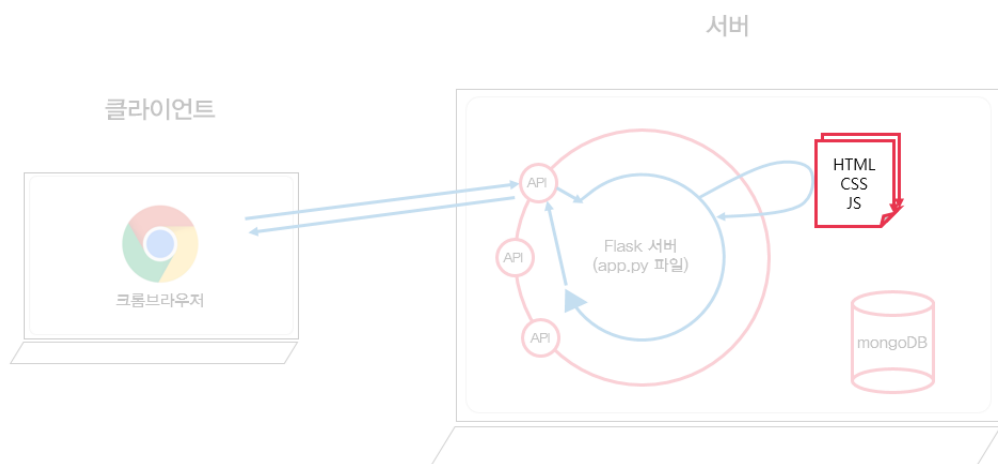
```
openapi.seoul.go.kr:8088/6d4d7 x +
< > ↻ 주의 요함 | openapi.seoul.go.kr:8088/6d4d776b466c656533356a4b4b5872/json/RealtimeCityAir/1/99

{
  - RealtimeCityAir: {
    list_total_count: 25,
    - RESULT: {
      CODE: "INF0-000",
      MESSAGE: "정상 처리되었습니다"
    },
    - row: [
      - {
        MSRDT: "202004241900",
        MSRRGN_NM: "도심권",
        MSRSTE_NM: "중구",
        PM10: 44,
        PM25: 20,
        O3: 0.039,
        NO2: 0.02,
        CO: 0.4,
        SO2: 0.003,
        IDEX_NM: "보통",
        IDEX_MVL: 59,
        ARPLT_MAIN: "PM10"
      },
      - {
        MSRDT: "202004241900",
```

#### ▼ 4. 웹 저장소

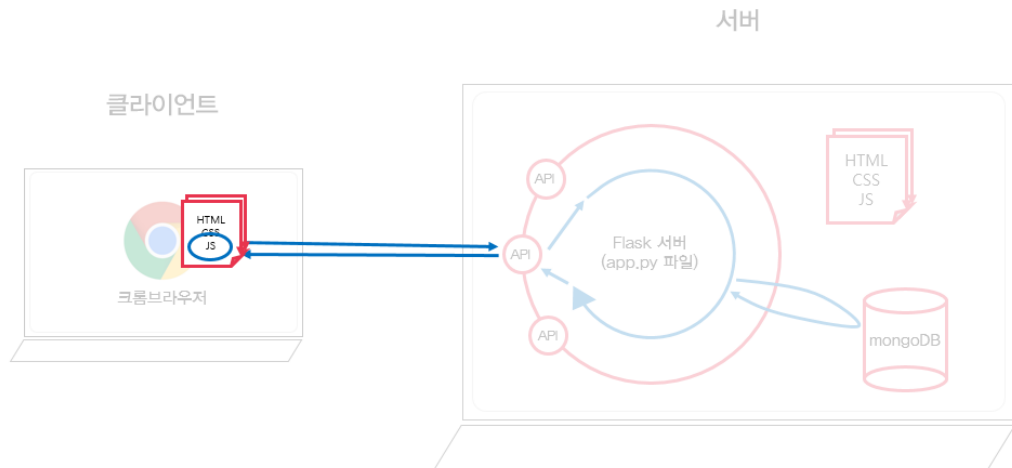
##### ▼ 1주차: HTML, CSS, Javascript

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



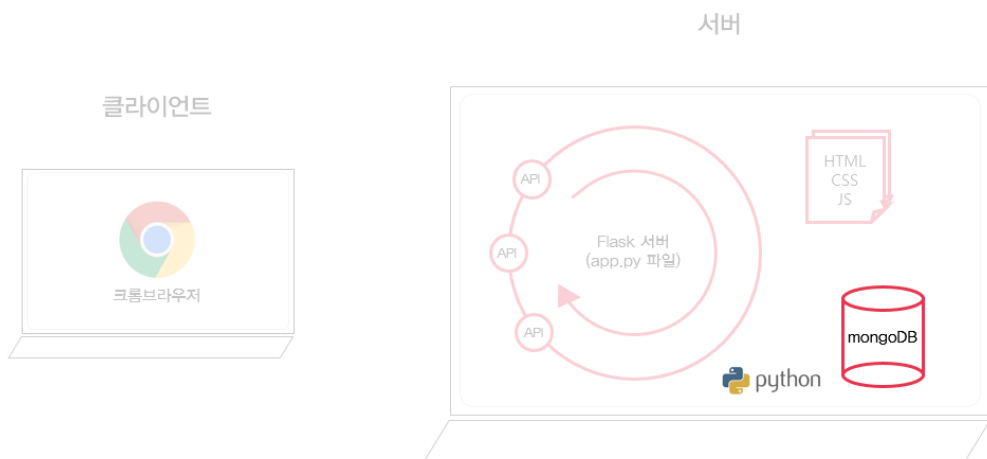
▼ 2주차: jQuery, Ajax, API

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



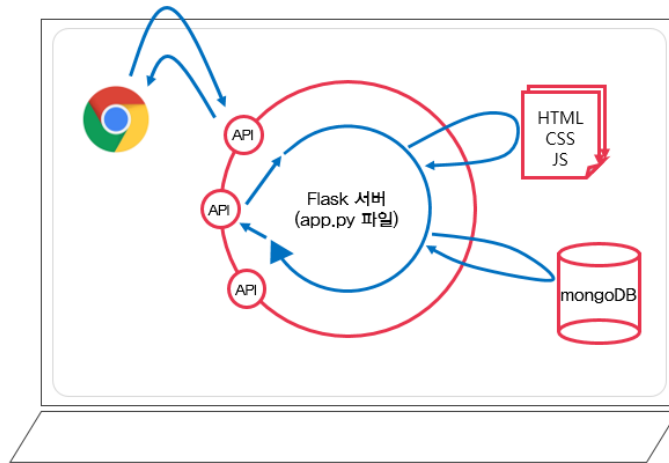
▼ 3주차: Python, 크롤링, mongoDB

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



▼ 4주차: 미니프로젝트1, 미니프로젝트2

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



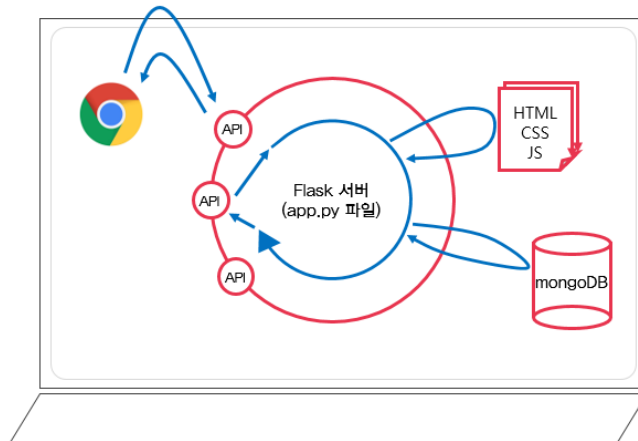
x 2개 더!



나중에 또 이야기하겠지만 헛갈리면 안되는 것!

우리는 컴퓨터가 한 대 잡아요... 그래서 같은 컴퓨터에다 서버도 만들고, 요청도 할 거예요. 즉, 클라이언트 = 서버가 되는 것이죠.

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



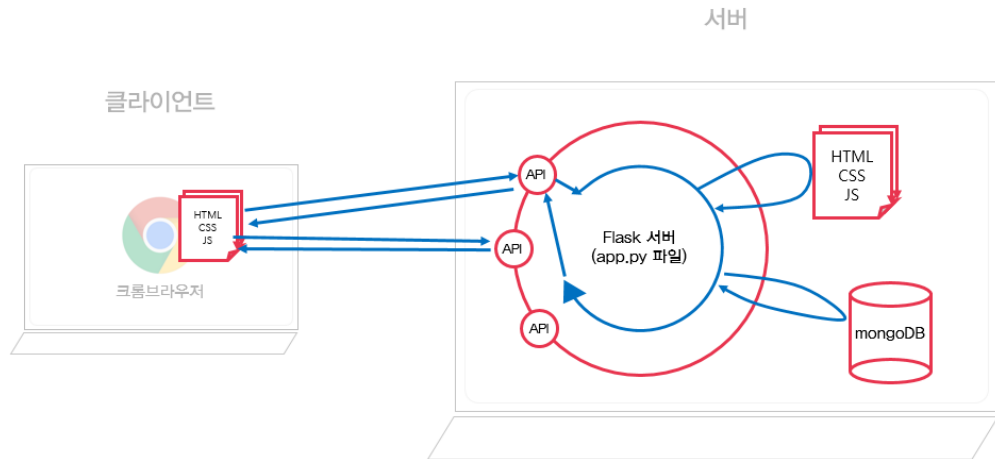
#### ▼ 5주차: 미니프로젝트3, AWS



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.



## ▼ 5. 인증

## 02. Hook - (1) Hook이란?

### ▼ 2) Hook이란?



Hook은 16.8버전부터 새로 추가된 기능입니다.

함수형 컴포넌트에서도 상태 값을 관리하거나 그 밖의 여러 React 기능을 사용할 수 있게 하기 위해 태어났어요.

- 혹은 자바스크립트 함수예요. 아래처럼 생겼습니다.

```
useEffect(() => {
  // something...
}, [])
```

- 혹은 사용할 때에도 규칙이 있습니다.
  - 1) 오직 함수형 컴포넌트에서만 쓸 수 있다.
    - 조금 더 정확히는 React Function 내에서만 쓸 수 있어요!
      - 리액트 함수는 리액트의 규칙을 따르는 함수정도로 생각해주세요. 함수형 컴포넌트와 커스텀 훅이 대표적입니다. (커스텀 훅은 커스텀 훅 만들기 때 조금 더 자세히 다뤄볼거예요. 😊)
  - 2) 컴포넌트의 최상위에서만 호출할 수 있다.
    - 반복문, 중첩문 내에서는 호출할 수 없어요.
      - 반복문, 중첩문 등에서 훅을 호출하게 될 경우, 컴포넌트가 렌더링할 때 훅의 실행 순서를 늘 동일하게 보장해줄 수 없기 때문입니다.



클래스형 컴포넌트가 있는데 굳이 훅을 추가하면서까지 함수형 컴포넌트를 만든 이유가 궁금하다면?

(공식 문서 보러가기 →)

### ▼ 3) 자주 써먹는 Hook - useState()

- useState()는 상태를 관리하기 위한 훅입니다.



#### 상태 관리

이미 앞서 배운 내용이지만, 중요한 개념이니까 다시 한 번 짚어볼게요.

컴포넌트가 리턴하는 리액트 엘리먼트는 불변 객체라고 말씀드린 적이 있지요?

이 불변 객체의 자식 노드 등 무언가를 바꿔주고 싶다면 어떤 값이 변했으니 어디를 업데이트하라고 해주어야 합니다. state는 여기서 어떤 값을 담당해요.

그리고 값이 변했다는 걸 명확히 알려주려면 임의로 값을 할당해서는 안됩니다.

정해진 규칙대로 값을 업데이트 해주어야 해요.



useState()는 값을 넣어두고 참조하기 위한 변수(state 변수), 값을 바꿔주기 위한 함수를 반환합니다.

```
const [someValue, setValue] = useState("hi!");
```



#### 구조 분해 할당

구조 분해 할당은 자바스크립트 표현식 중 하나입니다.

배열이나 객체의 속성을 해체해서 각각의 값을 개별 변수에 할당하게 해주는 거예요.

배열이나 딕셔너리의 []나 {}를 제거하고 안에 담긴 걸 새 변수에 담는다고 생각하면 편합니다.

위의 예시 코드에서는,

const [someValue, setValue] = useState("hi!"); 에서 useState는 [a, b]를 리턴해주고 있고, a와 b를 한 번에 가져오기 위해 [someValue, setValue]로 받아오는 거예요.

## 03. Hook - (2) 자주 써먹는 Hook

### ▼ 4) 자주 써먹는 Hook - useEffect()

- useEffect()는 컴포넌트의 사이드 이펙트 관리를 위한 훅입니다.





컴포넌트가 화면에 처음 그려진 후, 화면에서 수정된 후, 화면에서 사라질 때 어떤 작업을 주로 할까요?

네! API에서 어떤 데이터를 가져오거나, 이벤트 리스너를 구독하거나 혹은 스피너를 띄워주거나 (이런 건 DOM 수정이죠!) 등등의 작업을 할거예요. 이런 모든 것을 사이드 이펙트라고 합니다. `useEffect()`는 그런 사이드 이펙트를 관리하기 위한 훅이에요.

```
useEffect(() => {
  if(is_loaded){
    window.alert("hi! im ready! ٩(๏͇̂๏͈̂๏͉̂)");
  }
  return () => {
    window.alert("bye!");
  }
}, [is_loaded]);
```

- 위에서 사용한 `useEffect`의 인자는 2개 입니다.
  - 첫번째 인자는 컴포넌트가 화면에 그려질 때(처음 그려질 때와 수정되어 다시 그려질 때 모두) 실행할 함수예요.
  - 두번째 인자는 디펜던시 어레이라고 불러요. 의존성 배열입니다. 배열에 넣은 어떤 값이 변했을 때 `useEffect`에 넘긴 첫번째 인자를 다시 실행합니다.
- `return () => {}` 부분을 `clean up`이라고 불러요. 컴포넌트가 화면에서 사라질 때 마지막으로 정리(청소)한다는 거죠! 이벤트 리스너 등을 구독해제할 때 주로 여기서 해요.

#### ▼ 5) 자주 써먹는 Hook - `useCallback()`

- `useCallback()`는 함수를 메모이제이션 하기 위한 훅입니다.



메모이제이션이란 쉽게 말해 메모리 어딘가에 저장해두고 두고두고 써먹겠다는 거예요. 함수형 컴포넌트가 리렌더링되면 컴포넌트 안에 선언해둔 함수나 인라인으로 작성한 함수를 다시 한 번 생성하게 돼요.

어떤 컴포넌트가 총 10번 렌더링된다면 그 안에 작성해둔 함수들도 10번 만들어지는거고 이건 메모리 관리에 그리 효율적이지 않습니다. 😞

`useCallback()`은 함수를 메모이제이션해서 여러번 만들지 않게 해줘요.

주로 **자식 컴포넌트에게 전달해주는 콜백 함수를 메모이제이션할 때** 씁니다.

```
const myNewFunction = useCallback(() => {
  console.log("hey!", need_update);
}, [need_update]);
```

- `useEffect`와 마찬가지로 인자는 2개입니다.
  - 첫번째 인자는 실행할 콜백 함수입니다.
  - 두번째 인자는 디펜던시 어레이라고 불리는 의존성 배열입니다. 배열에 넣은 어떤 값이 변했을 때 콜백함수를 새로 생성하고 다시 메모이제이션합니다.



useCallback()은 함수를 최적화하기 위한 방법 중 하나예요.

보통 React.memo와 함께 사용해서 불필요한 렌더링을 방지하기 위해 씁니다. 오늘 맛보기로 배워보았지만 useCallback()을 사용한다고 해서 무조건 렌더링 횟수가 줄어들지 않습니다. 오히려 무분별하게 메모이제이션하는 건 좋지 않아요.

주로 자식 컴포넌트에게 props로 넘기는 콜백함수를 감싸는 데에 씁니다.

#### ▼ 6) 자주 써먹는 Hook - useRef()

- useRef()는 ref객체를 다루기 위한 훅입니다.



useRef()는 쉽게 설명하면 도플갱어 박스예요.

어떤 값을 넣어주면 그 값으로 초기화된 변경 가능한 ref 객체를 반환해주거든요!

그리고 이 값은 원본이 아니라 똑같이 생긴 다른 값이라 변경도 됩니다.

변경한다고 해도 리렌더링은 일어나지 않습니다.

```
const Input = () => {
  const input_ref = React.useRef(null);

  const clicker = (input_ref) => {
    console.log(input_ref);
  }

  return (
    <>
      <input ref={input_ref}/>
      <button onClick={clicker}>button</button>
    </>
  );
}
```

- .current
  - useRef는 .current 안에 값을 담아줘요.
  - 이 값을 변경해도 리렌더링은 일어나지 않습니다.
- useRef()는 주로 화면에 이미 노출된 것들을 관리할 때 사용합니다.
  - 인풋을 다루거나(포커스를 다루거나, 변경된 텍스트를 받아와야 하거나 등)
  - 버튼 클릭 후 등, 특정 상황에서 어떤 인터랙션을 보여주고 싶을 때
- DOM만을 위한 것은 아닙니다.
  - 엘리먼트 외에도 특정 값을 다루기 위해 사용할 수 있어요.

## 04. Quiz\_ 텍스트 입력기를 만들어보자!

- ▼ 7) 🖋️ 우측 인풋에 텍스트를 입력하고 [완성]버튼을 누르면 좌측 네모 박스에 입력한 텍스트가 나오게 해보기

- ▼ Q. 퀴즈설명: 그간 배운 내용을 바탕으로 텍스트 입력기를 만들어봅시다!



힌트:

1. 좌측 네모 박스, 우측 인풋, [완성]버튼을 각기 다른 컴포넌트로 분리해서 만들어봅시다.
2. 텍스트 값을 가져올 땐 `useRef()`를 사용하세요.
3. `useState()`를 사용해서 네모 박스에 값을 넣어주세요.
4. `useState()`는 App 컴포넌트에서 만들어주세요.

## ▼ A. 함께하기(완성분)

### ▼ [코드스니펫] - components.js

```
import React from "react";

export const TextArea = ({ text }) => {
  return (
    <div style={{ width: "50vw", border: "1px solid #888", minHeight: "20vh" }}>
      <pre>{text}</pre>
    </div>
  );
};

export const Button = ({ input_ref, setText }) => {
  return (
    <button
      onClick={() => {
        // console.log(input_ref.current.value);
        setText(input_ref.current.value);
        input_ref.current.value = "";
      }}
    >
      완성
    </button>
  );
};

export const Input = ({ input_ref }) => {
  return <input ref={input_ref} />;
};
```

### ▼ [코드스니펫] - App.js

```
import "../App.css";

import React from "react";

import { TextArea, Button, Input } from "../components";

function App() {
  const [text, setText] = React.useState("");
  const input_ref = React.useRef(null);
  return (
    <div className="App" style={{ display: "flex", gap: 10 }}>
      <div>
        <TextArea text={text} />
      </div>
      <div>
        <Input input_ref={input_ref} />
        <Button setText={setText} input_ref={input_ref} />
      </div>
    </div>
  );
}
```

```
export default App;
```

## 05. Custom Hook

### ▼ 8) [완성] 버튼 컴포넌트가 여러개가 된다면?

- 퀴즈에서 만든 텍스트 입력기에서 텍스트 영역에 텍스트를 넣어주는 또 다른 컴포넌트가 생긴다고 생각 해봅시다.



다른 컴포넌트가 생긴다면 ref를 가져오고 state에 값을 넣어주는 코드를 다시 작성해야겠죠. 이렇게 반복적인 로직이 여러개가 될 때는 해당하는 로직을 묶어서 하나의 함수로 빼놓으면 편합니다. 여러번 같은 코드를 작성할 필요 없이 함수 하나만 불러다 쓰면 되니까요!

### ▼ 9) custom hook이란?

- 반복되는 로직을 재사용하는 방법 중 하나입니다.



자바스크립트 함수에서는 반복되는 코드를 효율적으로 관리하기 위해 또 다른 함수로 묶어서 사용하곤 합니다. 커스텀 혹은 컴포넌트 내에서 혹은 사용하는 로직을 따로 분리하기 위한 거예요. 😊 (컴포넌트도 훅도 함수니까요!)

### ▼ 10) useCompletes 훅 만들기

- useCompletes 훅 만들기



custom hook도 결국 함수입니다. 리액트 함수지요. 일반 자바스크립트 함수처럼 막 만들 수는 없고, 약간의 규칙이 있어요.

1. 함수 명이 **use**로 시작해야 한다.
2. 최상위에서만 호출할 수 있다.
3. 리액트 함수 내에서만 호출할 수 있다.
4. 꼭 return 값을 주자.

```
//useCompletes.js
import React from "react";

export const useCompletes = (initial) => {
  const [text, setText] = React.useState(initial);

  const setBoxText = (_ref) => {
    const value = _ref.current?.value;
    if (value && value !== "") {
      setText(value);
      _ref.current.value = "";
    }
  };
};
```

```
    return [text, setBoxText];
  };

```

- 커스텀 혹은 아래처럼 사용할 수 있습니다.

```
//App.js
import './App.css';

import React from 'react';

import { TextArea, Button, Input } from './components';
import { useCompletes } from './useCompletes';

function App() {
  const input_ref = React.useRef(null);
  const [text, setText] = useCompletes("");

  return (
    <div className="App" style={{ display: "flex", gap: 10 }}>
      <div>
        <TextArea text={text} />
      </div>
      <div>
        <Input input_ref={input_ref} />
        <Button setText={setText} input_ref={input_ref} />
      </div>
    </div>
  );
}

export default App;

```

```
// components.js
import React from 'react';

export const TextArea = ({text}) => {
  return (
    <div style={{ width: "50vw", border: "1px solid #888", minHeight: "20vh" }}>
      <pre>{text}</pre>
    </div>
  );
};

export const Button = ({ input_ref, setText }) => {
  return (
    <button
      onClick={() => {
        setText(input_ref);
      }}
    >
      완성
    </button>
  );
};

export const Input = ({ input_ref }) => {
  return <input ref={input_ref} />;
};

```

## 06. 자바스크립트와 동시성

## ▼ 11) 자바스크립트와 쓰레드



자바스크립트는 싱글 쓰레드로 동작하는 언어입니다. (메인 쓰레드 하나와 콜스택 하나로 구성되어 있어요!)

그리고 비동기 작업을 동시에 할 수 있어요.

오잉? 😕 1번에 1개의 작업만 할 수 있는데, 어떻게 동시 실행을 할까요?

→ 자바스크립트는 코어 엔진만 가지고 돌아가지 않아요!

실행환경(런타임)의 도움을 받아 동시 실행을 합니다.

(WebAPI(dom, ajax, setTimeout...), Event Queue, Event Loop 등과 함께 동작합니다.)

- 쓰레드가 하나일 때 일어나는 일



아래 코드를 써보면 그 아래에 어떤 코드를 넣어두었더라도 멈춰있습니다. 일꾼이 하나니까 alert을 띄워주는 동안 다른 일을 못하는거예요.

+) 깊게 들어가면 이건 자바스크립트의 특징 중 하나인 run-to-completion이라는 것 때문인데요. 일 하나 끝내기 전엔 다른 건 하지 않는 거예요.

```
window.alert();  
console.log("hi!");
```

- 그런데 어떻게 비동기 작업을 하나?



**비밀은 이벤트 루프!** 😊

프론트엔드에서 자바스크립트는 혼자 독립 실행되는 게 아니고 브라우저를 통해 실행됩니다. 브라우저에서 자바스크립트를 실행할 때 이벤트 루프라는 걸 기반으로 실행하는데요, 이 이벤트 루프의 동작에 대해 알아보시다.

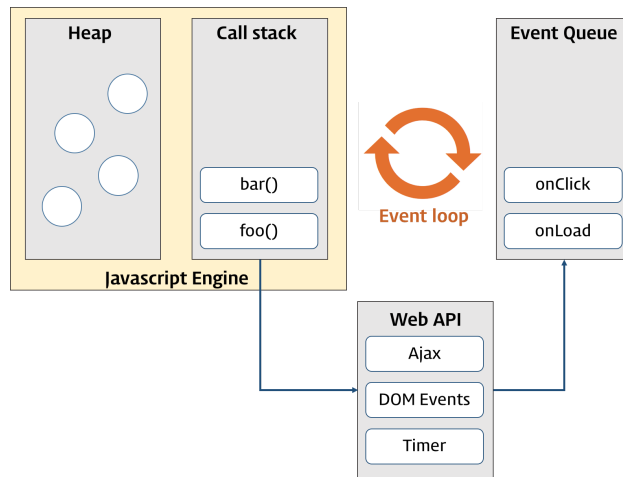
## ▼ 12) V8 엔진에서의 비동기 작업 처리



**동시성(Concurrency)**

자바스크립트는 기본적으로 한번에 하나의 일만 처리하지만, 우리가 만든 프로젝트가 돌아가는 걸 보면 여러 작업이 한 번에 처리되는 것처럼 느껴지죠!

이걸 두고 동시성이라고 부릅니다. 한 방에 여러개가 처리되는 것처럼 보이는 거요! (실제로 동시에 실행되는 건 아니예요. 실제로 동시에 실행되는 건 병렬이라고 불러요. 😊 )



- 용어정리
  - heap : 동적으로 생성된 객체 인스턴스가 할당되는 영역
  - call stack : 일거리가 쌓이는 스택
  - event queue : 테스크 큐(Task queue)나 콜백 큐(callback queue)라고도 해요. 비동기 처리 함수의 콜백 함수, 비동기식 이벤트 핸들러, 타이머의 콜백 함수를 넣어두는 큐
  - event loop : 테스크(일거리)가 들어오길 기다렸다가 테스크가 들어오면 일을 하고, 일이 없으면 잠깐 쉬기를 반복하는 자바스크립트 내의 루프
    - call stack 내에서 현재 실행중인 일거리가 있는 지, 이벤트 큐에 일거리가 있는 지 반복해서 확인하고, 콜 스택이 비어 있으면 이벤트 큐의 일거리를 콜스택으로 옮겨가게끔 도와요.
  - Web API : Ajax, DOM event, setTimeout 등 브라우저에 내장된 API

## 07. Promise

### ▼ 13) 동기와 비동기

- 동기

☞ 하나 하나씩 처리되는 방식을 동기라고 해요.  
 뭘 해줘!하고 요청을 보내고 다했어! 라는 응답을 받은 후에 다음 일을 시작하는 방식입니다.

- 비동기

☞ 어떤 작업을 해!하고 요청을 보낸 다음, 그 작업을 다했다는 응답을 받을 때까지 기다리지 않고 바로 다음 일을 시작하는 방식입니다.  
 서버 등에서 데이터를 받아오는 등 시간이 좀 필요한 작업을 해야한다면, 응답을 받기까지 계속 기다릴 수 없잖아요. 😓 그럴 때 비동기 처리를 합니다.

### ▼ 14) 콜백이란?



callback은 특정 함수에 매개변수로 전달된 함수예요.  
A()가 B()를 콜백으로 받았다면 A()안에서 B를 실행할 수 있을거예요.



콜백 패턴은 자바스크립트가 비동기 처리를 하기 위한 패턴 중 하나입니다!  
전통적인 콜백 패턴은 일명 콜백 헬로 불리는 엄청난 중첩 문제가 생기기 쉽습니다.

## 1. 콜백 헬

꼬리에 꼬리를 무는 비동기 처리가 늘어나면 호출이 계속 중첩되고, 코드가 깊어지고, 관리는 어려워집니다. 이런 깊은 중첩을 **콜백 헬**이나 **멸망의 피라미드**라고 부릅니다.

```
function async1('a', function (err, async2){
  if(err){
    errHandler(err);
  }else{
    ...
    async2('b', function (err, async3){
      ...
    }){
      ...
    }
  }
});
```

- 이런 콜백 헬이 발생하는 이유?
  - 비동기 처리 시에는 실행 완료를 기다리지 않고 바로 다음 작업을 실행해요.
  - 즉, 순서대로 코드를 짚 적는다고 우리가 원하는 순서로 작업이 이뤄지지 않습니다.
  - 비동기 처리 함수 내에서 처리 결과를 반환하는 걸로는 원하는 동작을 하지 않으니, 콜백 함수를 사용해 원하는 동작을 하게 하려고 콜백 함수를 씁니다.
  - 이 콜백 함수 내에서 또 다른 비동기 작업이 필요할 경우 위와 같은 중첩이 생기면서 콜백 헬이 탄생하죠. 😞

## ▼ 15) 프라미스란?



비동기 연산이 종료된 이후 결과를 알기 위해 사용하는 객체입니다!  
프라미스를 쓰면 비동기 메서드를 마치 동기 메서드처럼 값을 반환할 수 있어요.  
전통적인 콜백 패턴으로 인한 콜백 헬 때문에 ES6에서 도입한 또다른 비동기 처리 패턴입니다.  
**비동기 처리 시점을 좀 더 명확하게 표현할 수 있어요!**  
([독스 보러가기](#) →)

## 1. 프라미스 생성

- 프라미스는 Promise 생성자 함수(new 키워드 기억하시죠!)를 통해 생성합니다.
- 비동기 작업을 수행할 콜백 함수를 인자로 전달받아서 사용합니다.

```
// 프라미스 객체를 만듭니다.
// 인자로는 (resolve, reject) => {} 이런 excutor 실행자(혹은 실행 함수라고 불러요.)를 받아요.
// 이 실행자는 비동기 작업이 끝나면 바로 두 가지 콜백 중 하나를 실행합니다.
```



```
// resolve: 작업이 성공한 경우 호출할 콜백
// reject: 작업이 실패한 경우 호출할 콜백
const promise = new Promise((resolve, reject) => {
  if(...){
    ...
    resolve("성공!");
  }else{
    ...
    reject("실패!");
  }
});
```

## 2. 프라미스의 상태값

- pending: 비동기 처리 수행 전(resolve, reject가 아직 호출되지 않음)
- fulfilled: 수행 성공(resolve가 호출된 상태)
- rejected: 수행 실패(reject가 호출된 상태)
- settled: 성공 or 실패(resolve나 reject가 호출된 상태)

## 3. 프라미스 후속 처리 메서드

- 프라미스로 구현된 비동기 함수는 프라미스 객체를 반환하죠!
- 프라미스로 구현된 비동기 함수를 호출하는 측에서는 이 프라미스 객체의 후속 처리 메서드를 통해 비동기 처리 결과(성공 결과나 에러메시지)를 받아서 처리해야 합니다.
- .then(성공 시, 실패 시)  
then의 첫 인자는 성공 시 실행, 두번째 인자는 실패 시 실행됩니다. (첫 번째 인자만 넘겨도 됩니다!)

```
// 프라미스를 하나 만들어 봅시다!
let promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("완료!"), 1000);
});

// resolve
promise.then(result => {
  console.log(result); // 완료!가 콘솔에 찍힐거예요.
}, error => {
  console.log(error); // 실행되지 않습니다.
});
```

```
// 프라미스를 하나 만들어 봅시다!
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("오류!")), 1000);
});

// reject
promise.then(result => {
  console.log(result); // 실행되지 않습니다.
}, error => {
  console.log(error); // Error: 오류!가 찍힐거예요.
});
```

- .catch(실패 시)

```
// 프라미스를 하나 만들어 봅시다!
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("오류!")), 1000);
});
```

```
});

promise.catch((error) => {console.log(error)});
```

#### ▼ 16) promise chaining(프라미스 체이닝)

1. 프라미스는 후속 처리 메서드를 체이닝해서 여러 개의 프라미스를 연결할 수 있습니다! (이걸로 콜백 헬을 해결할 수 있어요!)

- 체이닝이 뭔데? 그걸 어떻게 하는 건데?

후속 처리 메서드 (then)을 쪽쪽 이어 주는 거예요.

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("promise 1"), 1000);
}).then((result) => { // 후속 처리 메서드 하나를 쓰고,
  console.log(result); // promise 1
  return new Promise(...);
}).then((result) => { // 이렇게 연달아 then을 써서 이어주는 거예요.
  console.log(result);
  return new Promise(...);
}).then(...);
```

## 08. async, await

#### ▼ 17) async, await



앞으로 정말정말 많이 보고 쓸 문법입니다.

프라미스 사용을 엄청 편하게 만들어줘요! 😊

#### [더 알면 좋은 내용]

강의에서 다루지 않지만 **제네레이터** 라는 함수와 **이터러블** 을 알면 더더 좋습니다.

#### 1. async

- 함수 앞에 async를 붙여서 사용합니다.
- 항상 프라미스를 반환합니다. (프라미스가 아닌 값이라도, 프라미스로 감싸서 반환해줘요!)

```
// async는 function 앞에 써줍니다.
async function myFunc() {
  return "프라미스를 반환해요!"; // 프라미스가 아닌 걸 반환해볼게요!
}

myFunc().then(result => {console.log(result)}); // 콘솔로 확인해봅시다!
```

#### 2. await

- async의 짝궁이에요. (async 없이는 못쓰입니다!)
- async 함수 안에서만 동작합니다.
- await는 프라미스가 처리될 때까지 기다렸다가 그 이후에 결과를 반환해요!

```
async function myFunc(){
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("완료!"), 1000);
```

```
});

console.log(promise);

let result = await promise; // 여기서 기다리자! 하고 신호를 줍니다.

console.log(promise);

console.log(result); // then(후처리 함수)를 쓰지 않았는데도, 1초 후에 완료!가 콘솔에 찍힐거예요.
}
```



await를 만나면, 실행이 잠시 중단되었다가 프라미스 처리 후에 실행을 재개합니다!  
즉, await를 쓰면 함수 실행을 기다리게 하는거예요.

## 09. 상태관리

### ▼ 18) 상태관리란?



상태관리는 쉽게 말해 전역 데이터 관리예요.  
렌더링과 연결된 데이터를 컴포넌트끼리 주고 받기는 생각보다 번거롭습니다.  
이런 번거로움을 줄여주기 위해 전역 데이터를 만들고 관리해주는 게 리액트에서 말하는 상태관리  
예요.

- 형제 컴포넌트끼리 데이터를 주고 받으려면?
  1. 부모 컴포넌트에서 state를 생성하고 자식 컴포넌트들에게 props로 데이터와 state를 변경할 함수를 넘겨준다.
  2. 자식 컴포넌트에서는 props로 받아온 값을 참조해서 쓰고, 값 변경이 필요한 경우는 넘겨받은 함수로 해당 값을 변경해준다.
  3. **[생각해보기]** 만약 1촌 관계 컴포넌트가 아니라 6촌 관계 컴포넌트가 같은 데이터를 사용하려고 한다면?
    - 으악! 생각만해도 끔찍하네요! props로 어디까지 넘겨줘야 하는건지... 🤔
- 많이 사용하는 상태관리 툴
  - ContextAPI()
  - Redux
  - Recoil
  - zustand
  - react-query
  - mobx

### ▼ 19) 전역 저장소는 내게 필요하지 않을 수 있다.



흔히 말하는 상태 관리는 어디서든 접근할 수 있는 데이터 모음을 만들어두고, 어떤 컴포넌트 건 데이터를 꺼내서 보고 수정 요청을 보낼 수 있는, 전역 저장소 개념입니다.  
내 프로젝트는 컴포넌트간 데이터가 오갈 일이 많지 않다면 정말x100 전역 저장소는 필요 없어요.  
모든 프로젝트에 상태관리가 필요하지 않다는 걸 꼭 기억해주세요.

## 10. 상태관리 해보기 - (1) ContextAPI()

### ▼ 20) 데이터를 저장할 공간부터 만들자!

- `React.createContext()`

```
const MyStore = React.createContext();
```

### ▼ 21) 데이터를 가져다 쓰려면?

- `Context.Provider`



데이터를 주입할 차례네요!

`Provider`는 `Context`를 구독한 컴포넌트들에게 앳, 나 지금 데이터 변했어! 하고 알려줍니다.  
여러 컨텍스트가 있다면 중첩해서 써도 괜찮아요. 😊

### ▼ Provider 예시

```
<MyStore.Provider value={state}>
  <Component1 />
  <Component2 />
</MyStore.Provider>
```

- `Context.Consumer`



주입한 데이터를 구독해봅시다!

`Consumer`는 **컴포넌트**가 `context`를 구독하게 해줍니다.

### ▼ Consumer 예시

```
function App() {
  // Context의 Value는 App 컴포넌트에서 관리하자!
  const [name, setName] = useState("민영");
  return (
    <MyStore.Provider value={{ name, setName }}>
      <MyStore.Consumer>
        {(value) => {return <div>{value.name}</div>}}
      </MyStore.Consumer>
    </MyStore.Provider>
  );
}
```

👉 Consumer는 위처럼 사용하지만 useContext()를 쓰면 굳이 쓰지 않아도 괜찮아요. 😊

#### ▼ useContext() 예시

```
// MyStore.Provider를 찾는데 성공할 컴포넌트
const SomeComponentInProvider = () => {
  const { name, setName } = useContext(MyStore);
  return (
    <div>
      {name}
      <button onClick={() => setData("perl")}>바꾸기</button>
    </div>
  );
}
```

#### ▼ 22) 데이터 수정하기

- 데이터 수정하기

👉 context를 만들 때, 값과 함수를 만들었죠?  
값은 가져다 쓰기 위함이고 함수는 값을 고쳐쓰기 위함입니다.  
만든 함수를 사용해 값을 고쳐봅시다.

```
import React, { useState, useContext } from "react";
// Context를 만들기
const MyStore = React.createContext();

const MyStoreConsumer = () => {
  const { name, setName } = useContext(MyStore);
  return (
    <div>
      {name}
      <button onClick={() => setName("perl")}>바꾸기</button>
    </div>
  );
};

function App() {
  // Context의 Value는 App 컴포넌트에서 관리
  const [name, setName] = useState("민영");
  return (
    <MyStore.Provider value={{ name, setName }}>
      <MyStoreConsumer />
    </MyStore.Provider>
  );
}

export default App;
```

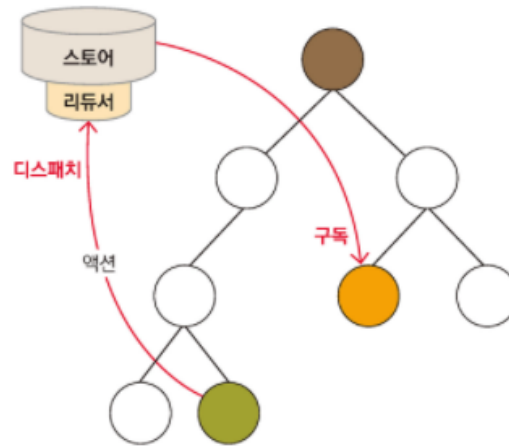
## 11. 상태관리 해보기 - (2) Redux 1

#### ▼ 23) 상태관리 흐름을 알아보자!



[상태관리 흐름도]

딱 4가지만 알면 됩니다! Store, Action, Reducer, 그리고 Component!  
아주 큰 흐름만 잘 파악해도 굳굳!



- (1) 리덕스 Store를 Component에 연결한다.
- (2) Component에서 상태 변화가 필요할 때 Action을 부른다.
- (3) Reducer를 통해서 새로운 상태 값을 만들고,
- (4) 새 상태값을 Store에 저장한다.
- (5) Component는 새로운 상태값을 받아온다. (props를 통해 받아오니까, 다시 랜더링 되겠죠?)

#### ▼ 24) redux 복습



리덕스는 아주 흔히 사용하는 **상태관리 라이브러리**입니다.

**전역 상태관리**를 편히 할 수 있게 해주는 고마운 친구죠!

[공식문서 보러가기](#) →

#### ▼ (1) State



리덕스에서는 저장하고 있는 상태값("데이터"라고 생각하셔도 돼요!)를 **state**라고 불러요.  
딕셔너리 형태(`{[key]: value}`)형태로 보관합니다.

#### ▼ (2) Action



상태에 변화가 필요할 때(=가지고 있는 데이터를 변경할 때) 발생하는 것입니다.

```
// 액션은 객체예요. 이런 식으로 쓰여요. type은 이름같은 거예요! 저희가 정하는 임의의 문자열을 넣습니다.
{type: 'CHANGE_STATE', data: {...}}
```

### ▼ (3) ActionCreator



액션 생성 함수라고도 부릅니다. 액션을 만들기 위해 사용합니다.

```
//이름 그대로 함수예요!  
const changeState = (new_data) => {  
  // 액션을 리턴합니다! (액션 생성 함수니까요. 제가 너무 당연한 이야기를 했나요? :))  
  return {  
    type: 'CHANGE_STATE',  
    data: new_data  
  }  
}
```

### ▼ (4) Reducer



리덕스에 저장된 상태(=데이터)를 변경하는 함수입니다.

우리가 액션 생성 함수를 부르고 → 액션을 만들면 → 리듀서가 현재 상태(=데이터)와 액션 객체를 받아서 → 새로운 데이터를 만들고 → 리턴해줍니다.

```
// 기본 상태값을 임의로 정해줬어요.  
const initialState = {  
  name: 'mean0'  
}  
  
function reducer(state = initialState, action) {  
  switch(action.type){  
  
    // action의 타입마다 케이스문을 걸어주면,  
    // 액션에 따라서 새로운 값을 돌려줍니다!  
    case CHANGE_STATE:  
      return {name: 'mean1'};  
  
    default:  
      return false;  
  }  
}
```

### ▼ (5) Store



우리 프로젝트에 리덕스를 적용하기 위해 만드는 거예요!

스토어에는 리듀서, 현재 애플리케이션 상태, 리덕스에서 값을 가져오고 액션을 호출하기 위한 몇 가지 내장 함수가 포함되어 있습니다.

생김새는 딕셔너리 혹은 json처럼 생겼어요.

내장함수를 어디서 보나요? → 공식문서에서요! 😊

### ▼ (6) dispatch



디스패치는 우리가 앞으로 정말 많이 쓸 스토어의 내장 함수예요!

액션을 발생 시키는 역할을 합니다.

```
// 실제로는 이것보다 코드가 길지만,  
// 간단히 표현하자면 이런 식으로 우리가 발생시키고자 하는 액션을 파라미터로 넘겨서 사용합니다.  
dispatch(action);
```

▼ 25) 나는 리덕스가 필요하지 않을 수 있다.



리덕스가 굉장히 무겁다는 이야기, 들어보셨나요?

"리덕스 사용이 과연 프로젝트를 나이스하게 만들어줄까?"에 관한 고민은 아주아주 오래 이어져왔습니다.

저는 리덕스를 아주 좋아하는 사람이라 가급적 리덕스를 사용하지만, 아래의 경우에 해당한다면 가차없이 다른 라이브러리를 택합니다.

1. 페이지 간 공유할 데이터가 없는 경우
2. 페이지 이동 시 리패칭이 잦게 일어날 경우
3. 비즈니스 로직이 획일화되기 어려울 경우

## 12. 상태관리 해보기 - (3) Redux 2



### Redux Tool Kit

리덕스툴킷은 리덕스를 편히 쓰라고 리덕스 팀에서 만든 패키지입니다.

리덕스를 사용하는데 유용한 패키지가 몽땅 들어있는 도구 모음이에요. 😊

툴 킷을 쓰게 되면 액션타입, 액션 생성함수, 리듀서, 기초값을 한 번에 묶어서 사용할 수 있어요.

→ 보일러 플레이트가 많이 줄어듭니다.

▼ 26) store 만들기

▼ import 할 것부터 다 해보자!

▼ [코드스니펫] import 하기

```
import { createStore, combineReducers, applyMiddleware, compose } from "redux";  
import thunk from "redux-thunk";  
import { createBrowserHistory } from "history";  
import { connectRouter } from "connected-react-router";  
  
import User from "../modules/user";
```

▼ root reducer 만들자!

▼ [코드스니펫] rootreducer 만들기

```
const rootReducer = combineReducers({  
  user: User,  
});
```

▼ 미들웨어도 준비하고,

▼ [코드스니펫] 미들웨어 준비



```
const middlewares = [thunk];

// 지금이 어느 환경인 지 알려줘요. (개발환경, 프로덕션(배포)환경 ...)
const env = process.env.NODE_ENV;

// 개발환경에서는 로거라는 걸 하나만 더 써볼게요.
if (env === "development") {
  const { logger } = require("redux-logger");
  middlewares.push(logger);
}
```

#### ▼ 크롬 확장 프로그램, redux devTools도 사용 설정하기



redux devTools 공식문서에 있는 사용방법대로 해볼거예요.  
더 알아보고 싶다면 공식문서를 봐주세요! ([링크](#))

#### ▼ [코드스니펫] redux devTools 설정

#### ▼ root reducer 만들자!

#### ▼ [코드스니펫] rootreducer 만들기

```
const rootReducer = combineReducers({
  user: User,
});
```

#### ▼ 미들웨어도 준비하고,

#### ▼ [코드스니펫] 미들웨어 준비

```
const middlewares = [thunk];

// 지금이 어느 환경인 지 알려줘요. (개발환경, 프로덕션(배포)환경 ...)
const env = process.env.NODE_ENV;

// 개발환경에서는 로거라는 걸 하나만 더 써볼게요.
if (env === "development") {
  const { logger } = require("redux-logger");
  middlewares.push(logger);
}
```

#### ▼ 크롬 확장 프로그램, redux devTools도 사용 설정하기



redux devTools 공식문서에 있는 사용방법대로 해볼거예요.  
더 알아보고 싶다면 공식문서를 봐주세요! ([링크](#))

#### ▼ [코드스니펫] redux devTools 설정

```
const composeEnhancers =
  typeof window === "object" && window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__
    ? window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__({
```

```
// Specify extension's options like name, actionsBlacklist, actionsCreators, serialize...
}))
: compose;
```

## ▼ 이제 미들웨어를 묶자!

### ▼ [코드스니펫] 미들웨어 묶기

```
const enhancer = composeEnhancers(
  applyMiddleware(...middlewares)
);
```

## ▼ 미들웨어하고 루트 리듀서를 엮어서 스토어를 만든다!

### ▼ [코드스니펫] 스토어 만들기

```
let store = (initialStore) => createStore(rootReducer, enhancer);

export default store();
```

## ▼ 27) slice 만들기

### ▼ import 부터!

```
import { createAction, handleActions } from "redux-actions";
import { produce } from "immer";
```

## ▼ 필요한 액션이 뭐가 있을 지 먼저 만들어주고,

```
const SET_POST = "SET_POST";
const ADD_POST = "ADD_POST";

const setPost = createAction(SET_POST, (post_list) => ({post_list}));
const addPost = createAction(ADD_POST, (post) => ({post}));
```

## ▼ initialState 만들기

```
const initialState = {
  list: [],
}

// 게시물 하나에는 어떤 정보가 있어야 하는 지 하나 만들어둬시다! :)
const initialPost = {
  user_info: {
    id: 0,
    user_name: "mean0",
    user_profile: "https://mean0images.s3.ap-northeast-2.amazonaws.com/4.jpeg",
  },
  image_url: "https://mean0images.s3.ap-northeast-2.amazonaws.com/4.jpeg",
  contents: "고양이네요!",
  comment_cnt: 10,
  insert_dt: "2021-02-27 10:00:00",
};
```

## ▼ 리듀서 작성하기

```
// reducer
export default handleActions(
  {
    [SET_POST]: (state, action) => produce(state, (draft) => {

    })),

    [ADD_POST]: (state, action) => produce(state, (draft) => {

    })
  },
  initialState
);
```

#### ▼ export 까지 하면 끝!

```
// action creator export
const actionCreators = {
  setPost,
  addPost,
};

export { actionCreators };
```

#### ▼ 28) redux hook으로 데이터 구독하기

```
// pages/PostList.js
...
import {useSelector} from "react-redux";
...
const PostList = (props) => {
  const post_list = useSelector((state) => state.post.list);
  ...
}
```

#### ▼ 29) redux hook으로 데이터 수정하기

```
...
import {actionCreators as userActions} from "../redux/modules/user";
import { useDispatch } from "react-redux";

const Login = (props) => {
  const dispatch = useDispatch();
  const [id, setId] = React.useState('');
  const [pwd, setPwd] = React.useState('');

  const changeId = (e) => {
    setId(e.target.value);
  }

  const changePwd = (e) => {
    setPwd(e.target.value);
  }

  const login = () => {
    dispatch(userActions.login({user_name: "perl"}));
  }

  return (
    <React.Fragment>
      <Grid padding={16}>
        <Text type="heading">로그인 페이지</Text>
      </Grid>
    </React.Fragment>
  );
}
```

```

    <Grid padding={16}>
      <Input value={id} onChange={changeId} placeholder="아이디를 입력하세요." />
      <Input value={pwd} onChange={changePwd} type="password" placeholder="비밀번호를 입력하세요." />
    </Grid>


    <Button __click={() => {login();}}>로그인</Button>
  </React.Fragment>
)
}

export default Login;

```

## 13. 라우팅


### ▼ 30) react-router-dom 설치하기

 강의는 v6 기준으로 진행됩니다. 설치하실 때 현재 패키지 버전을 확인하시고 강의와 버전을 맞춰주세요. 😊  
**공식문서 보러가기** →

- 먼저 새프로젝트를 만들어주세요! ( ⇒ Route\_ex 프로젝트 만들기!)
- react-router-dom 설치

```
yarn add react-router-dom
```

### ▼ 31) 라우팅이란?

 라우팅은 페이지 전환이에요.  
SPA 방식에서는 첫 로딩 때 모든 정적 자원(js, css 등)을 한 번에 가져오죠!  
MPA 방식에서는 주소에 맞는 html을 열어주었지만 SPA는 주소창을 보고 주소에 맞게 매번 페이지를 조립해서 보여주어야 합니다.  
우리는 react-router-dom 패키지를 사용해서 해볼거예요.

### ▼ 32) 라우팅 해보기

#### ▼ (1) App.js에 BrowserRouter 적용하기

- BrowserRouter 적용하기

```

import React from "react";
import ReactDOM from "react-dom";
import { BrowserRouter } from "react-router-dom";

function App() {
  return <h1>Hello React Router</h1>;
}

ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,

```

```
document.getElementById("root")
);
```



BrowserRouter(브라우저라우터)는 웹 브라우저가 가지고 있는 주소 관련 정보를 props로 넘겨주는 친구입니다.

현재 내가 어느 주소를 보고 있는 지 쉽게 알 수 있게 도와줘요.

## ▼ (2) 세부 화면 만들기

### • Home.js

```
import React from "react";

const Home = (props) => {

  return (
    <div>메인 화면이에요.</div>
  )
}

export default Home;
```

### • Cat.js

```
import React from "react";

const Cat = (props) => {

  return (
    <div>고양이 화면이에요.</div>
  )
}

export default Cat;
```

### • Dog.js

```
import React from "react";

const Dog = (props) => {

  return (
    <div>강아지 화면이에요.</div>
  )
}

export default Dog;
```

## ▼ (3) App.js에서 Route 적용하기

### • Route 사용방법 1: 넘겨줄 props가 없을 때

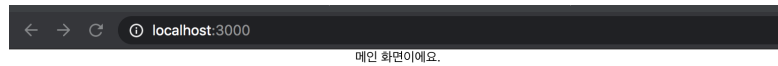
```
<Route path="/cat">
  <Cat />
</Route>
```

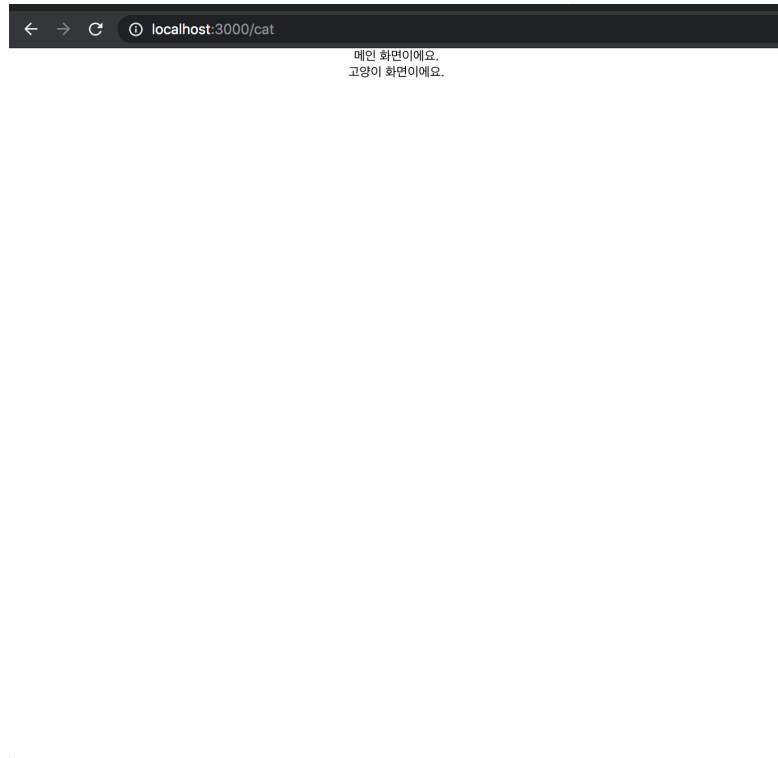
- Route 사용방법 2: 넘겨줄 props가 있을 때

```
<Route
  path="/cat"
  element={<Cat data="전달된 props" />}
/>
```

- App.js에 적용해보자

- 주소창에 "/", "/cat", "/dog"을 입력해보자!





주소에 /를 입력했을 때는 Home컴포넌트만 뜨는데 왜 /cat에서는 Home과 Cat이 다 뜨는걸까요? 정답은 다음 항목에 있어요!

#### ▼ (4) URL 파라미터사용하기

- 웹사이트 주소에는 파라미터와 쿼리라는 게 있어요. 우리는 그 중 파라미터 사용법을 알아볼 거예요!
- 이렇게 생겼어요!
  - 파라미터: /cat/nabi
  - 쿼리: /cat?name=nabi
- 파라미터 주는 방법

```
//App.js
...
// 파라미터 주기
<Route path="/cat/:cat_name" element={Cat}/>
...

```

- 파라미터 사용 방법

```
//Cat.js
import React from "react";

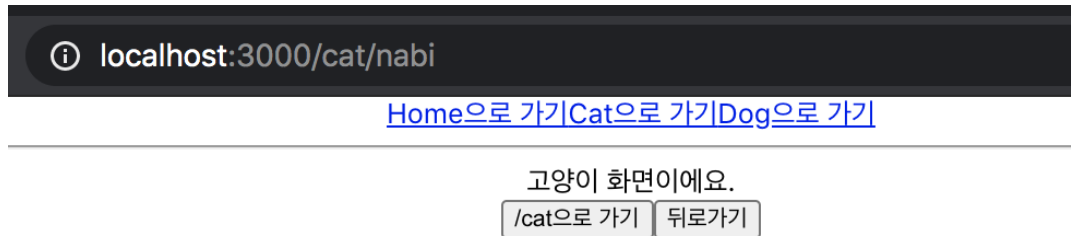
const Cat = (props) => {

  console.log(props.match);

  return (
    <div>고양이 화면이에요.</div>
  )
}
```

```
}
export default Cat;
```

- /cat/nabi로 주소를 이동해서 콘솔에 파라미터가 어떻게 찍히나 확인해봅시다!



```
▼ {path: "/cat/:cat_name", url: "/cat/nabi", isExact: true, params: {...}} ⓘ Cat.js:5
  isExact: true
  ▶ params: {cat_name: "nabi"}
  path: "/cat/:cat_name"
  url: "/cat/nabi"
  ▶ __proto__: Object
```

#### ▼ useParams 혹은 사용해서 이동하기



꼭 props에서 받아오지 않아도, useParams 혹은 사용하면 간단히 파라미터에 접근할 수 있어요! 엄청 편하겠죠. 🥰

```
import React from "react";
import { useParams } from "react-router-dom";
const Cat = (props) => {
  const cat_name = useParams();
  console.log(cat_name);
  // console.log(props);
  return (
    <div>고양이 화면입니다!</div>
  );
};
export default Cat;
```

#### ▼ (5) 링크 이동 시키기



매번 주소창을 찍고 페이지를 돌아다닐 순 없겠죠! react-router-dom으로 페이지를 이동하는 방법을 알아봅시다!

#### ▼ -1) <Link/> 사용하기

- <Link/> 사용 방법

링크 컴포넌트는 html 중 a 태그와 비슷한 역할을 해요. 리액트 내에서 페이지 전환을 도와줍니다.



```
<Link to="주소">[텍스트]</Link>
```

- App.js에 메뉴를 넣어보자!

→ 우리가 만든 메뉴처럼 <route> 바깥에 있는 돔요소는 페이지가 전환되어도 그대로 유지됩니다. (편리하죠!)

```
// Route를 먼저 불러와줍니다.
// Link 컴포넌트도 불러왔어요.
import { Route, Link } from "react-router-dom";

// 세부 페이지가 되어줄 컴포넌트들도 불러와주고요!
import Home from "./Home";
import Cat from "./Cat";
import Dog from "./Dog";

function App() {
  return (
    <div className="App">
      <div>
        <Link to="/">Home으로 가기</Link>
        <Link to="/cat">Cat으로 가기</Link>
        <Link to="/dog">Dog으로 가기</Link>
      </div>
      { /* 실제로 연결해볼까요! */ }
      <Route path="/" exact>
        <Home />
      </Route>
      <Route path="/cat" component={Cat}>
        { /* <Cat /> */ }
      </Route>
      <Route path="/dog">
        <Dog />
      </Route>
    </div>
  );
}

export default App;
```

## ▼ -2) navigate 사용하기

### ▼ useNavigate 혹은 사용해서 이동하기

```
import { useNavigate } from "react-router-dom";

function App() {
  let navigate = useNavigate();
  function handleClick() {
    navigate("/home");
  }
  return (
    <div>
      <button onClick={handleClick}>go home</button>
    </div>
  );
}
```

## 14. 끝 & 숙제 설명



이번 주차에는 효과 상태관리, 라우팅을 배웠어요! 비동기에 관련한 이론도 많이 배웠습니다. 이 내용을 바탕으로 숙제를 풀어봐요!

1. 페이지를 나눠봅시다.
2. /write에서 [추가하기]를 누르면 /에도 추가되도록 리덕스를 붙여봅시다.

### ▼ 예시 화면

- /write

- /

TIL

추가

[https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b331f5e7-94af-459f-8355-793bad7de45c/homework\\_w2.zip](https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b331f5e7-94af-459f-8355-793bad7de45c/homework_w2.zip)

Copyright © TeamSparta All rights reserved.