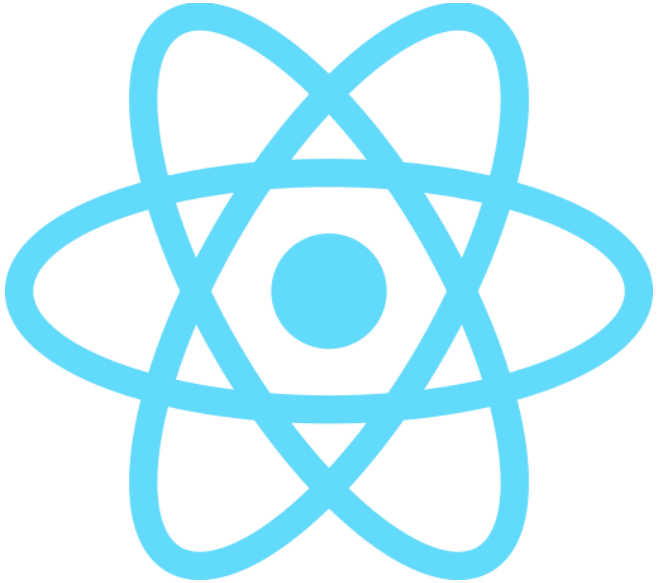


React

With Next.js

2. State & Hook

연사 : 박승범 (@killerwhale)

연사 & 멘토 소개

연사 : 박승범 (@killerwhale)

멘토 : 박승범 (@killerwhale)

멘토 : 박승범 (@killerwhale)

컴포넌트의 상태

Selected = True

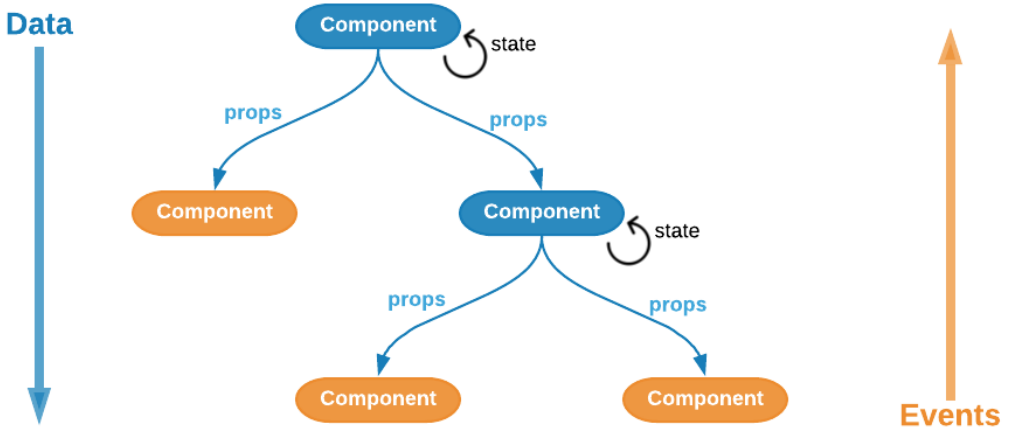
Click!

Click!

Selected = False

Selected에 따라 다른 UI의 컴포넌트 렌더링
-> JSX를 활용해서 가능

Selected 라는 값 (상태)를 변경하려면?
-> React의 훅을 활용한다.



컴포넌트의 생명 주기

컴포넌트의 Mount, Update, Unmount 시점 정하기

Mount : 컴포넌트가 첨음으로 Dom 트리에 삽입되는 시점
(컴포넌트가 처음 렌더링 될 때)

Update : 컴포넌트의 상태가 바뀌는 등 변화로 인해
컴포넌트에 변화가 생길 때

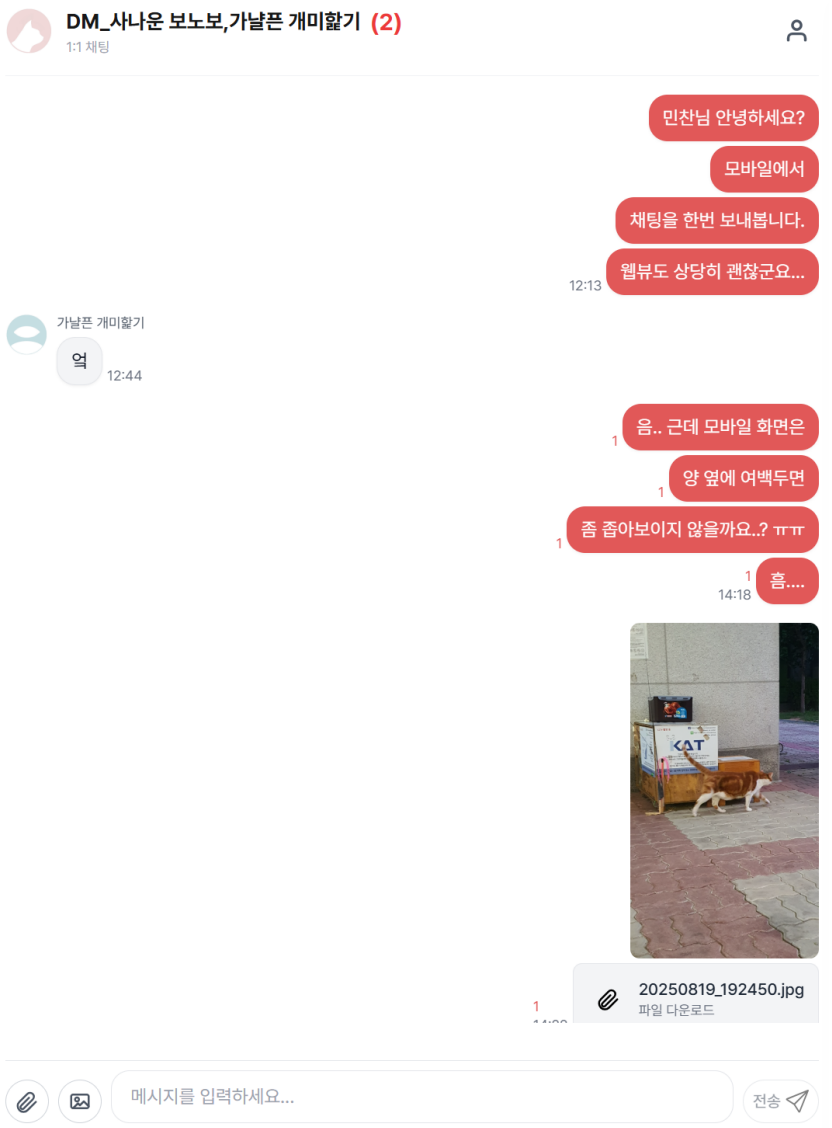
Unmount : 컴포넌트가 완전히 정리될 때

<채팅방 컴포넌트의 생명주기>

Mount : 채팅방 페이지에 들어간 경우

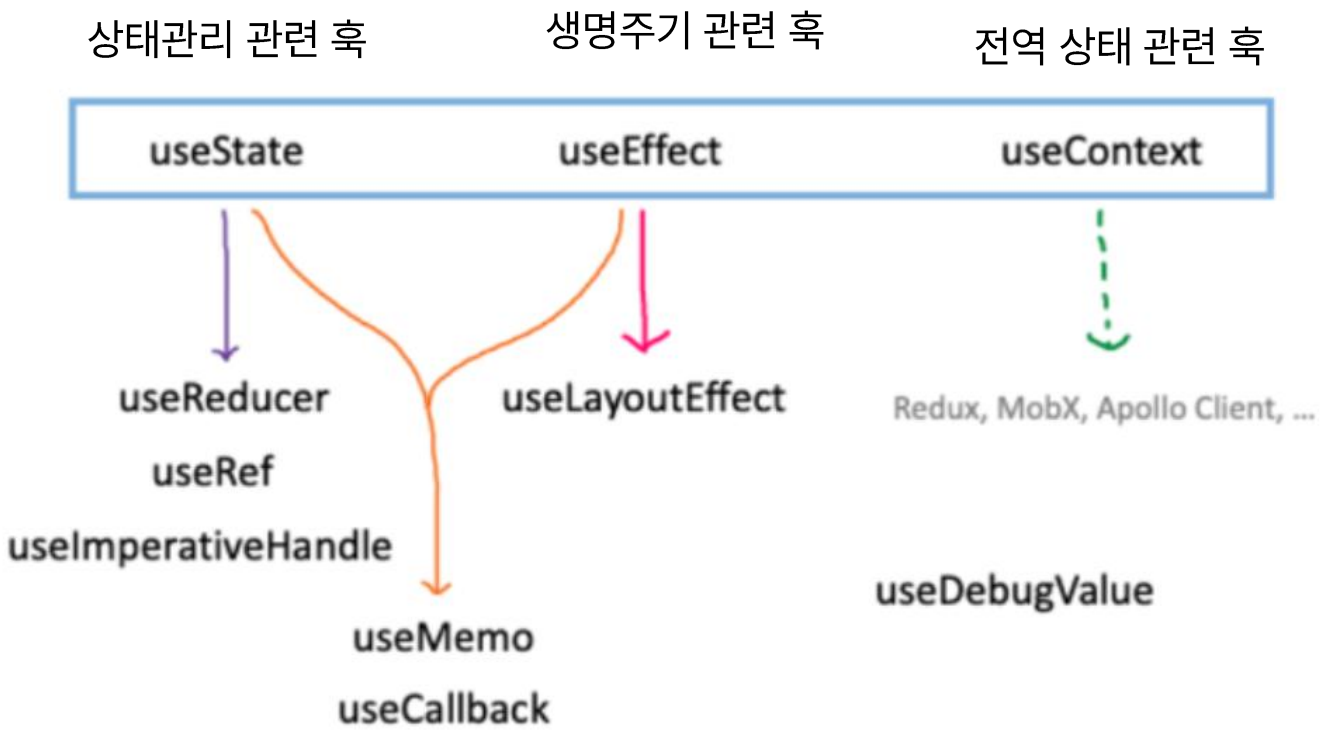
Update : 나 또는 상대방이 메시지를 보냈을 때 (or 삭제했을 때)

Unmount : 채팅방 페이지에서 나갔을 때



Hook

React의 FC (Functional Component)에서 **상태관리**, **생명주기** 관련 기능을 활용하기 위한 도구



Rule of Hook

React의 Hook의 이름은 항상 'use' 로 시작해야 한다.

Hook은 컴포넌트 내부에서만 정의할 수 있으며, 항상 최상단에 정의한다. (Hook의 Side Effect 제어를 위함)

```jsx
import React, { useState, useEffect } from "react";

function Counter() {
  // ✅ 최상위 레벨에서 훅을 호출 → 항상 같은 순서로 실행됨
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("컴포넌트가 렌더링됨");
  }, []);

  return (
    <div>
      <p>현재 값: {count}</p>
      <button onClick={() => setCount(count + 1)}>증가</button>
    </div>
  );
}

export default Counter;
```

```jsx
function Counter({ show }) {
  if (show) {
    // ❌ 조건문 안에서 훅 호출 → 렌더링마다 실행 순서가 달라질 수 있음
    const [count, setCount] = useState(0);
  }

  return <div>잘못된 코드</div>;
}
```

```jsx
function BadComponent() {
  function doSomething() {
    // ❌ 일반 함수 안에서 훅 호출
    const [count, setCount] = useState(0);
  }

  return <button onClick={doSomething}>Click</button>;
}
```

Mount -> Update -> Unmount 이외 시점은 컨트롤 X

React에서 제공하는 Hook 이외에 필요할 경우 커스텀 훅을 만들 수 있다.

useState

컴포넌트 내부의 상태 변수를 정의하기 위한 Hook

```jsx
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount((count) => count + 1)}>
      count is {count}
    </button>
  )
}
```

초기 값

상태의 현재 값과, 상태를 업데이트하는 함수가 반환됨

상태를 업데이트할 때에는 **반드시** 제공된 업데이트 함수를 사용해야 함

count : getter / setCount : setter 의 개념으로 이해하면 된다.

setCount 호출시 -> 컴포넌트를 다시 렌더링 한다. (자식 컴포넌트도 함께 rerendering)

```jsx
const [boardConfigs, setBoardConfigs] = useState<BoardConfig[]>([]);
const [selectedBoard, setSelectedBoard] = useState<BoardConfig | null>(null);
const [currentBoardType, setCurrentBoardType] = useState<'all' | 'popular' | 'board'>('all');
const [currentBoardId, setCurrentBoardId] = useState<number | null>(null);

const [selectedTopicId, setSelectedTopicId] = useState<string>('');
const [search, setSearch] = useState<string>('');
const [searchInput, setSearchInput] = useState<string>('');
```

한 컴포넌트 내에서 여러 개의 State 지정이 가능하고, 각각의 타입을 <>로 지정할 수 있다.

useParams / usePathname / useSearchParams

useParams : /chat/[id] 와 같은 Next의 동적 라우팅 에서 동적 경로 파라미터를 가져오는 훅

```tsx
export default function ChatRoomPage() {
  const params = useParams<{ room_id: string }>();
  const roomId = useMemo(() => {
```

Chat/[room_id] 라우팅 – chat/123 에서
Params 에는 { room_id : “123” } 이 저장된다.

usePathname : 현재 url을 string으로 가져오는 훅

```tsx
// URL: /blog/123?sort=asc
const pathname = usePathname(); // "/blog/123"
```

useSearchParams : queryString의 상태 관리 Hook

```tsx
"use client";
import { useSearchParams } from "next/navigation";

export default function Example() {
  const searchParams = useSearchParams();
  const sort = searchParams.get("sort"); // ?sort=asc → "asc"

  return <p>정렬: {sort}</p>;
}
```

세 훅은 각각
동적 경로 파라미터, url, 쿼리스트링이 변화할 때
컴포넌트를 다시 렌더링 한다.

useEffect

컴포넌트의 상태 변화에 따른 task를 등록하기 위한 Hook

```tsx
export default function ChatRoomPage() {
  const params = useParams<{ room_id: string }>();
  const roomId = useMemo(() => {
    const id = Array.isArray(params?.room_id) ? params.room_id[0] : params?.room_id;
    return id ? parseInt(id, 10) : null;
  }, [params]);

  useEffect(() => {
    fetchChatRoomList()
      .then((data) => {
        const sortedRooms = [...(data.results || [])].sort((a, b) => {
          const aTime = new Date(a.recent_message_at || a.created_at || 0).getTime();
          const bTime = new Date(b.recent_message_at || b.created_at || 0).getTime();
          return bTime - aTime;
        });
        setRooms(sortedRooms);

        // 현재 roomId에 해당하는 방 정보 찾기
        if (roomId) {
          const room = sortedRooms.find(r => r.id === roomId);
          setCurrentRoom(room);
        }
      });
  }, [roomId]);
```

컴포넌트가 Mount될 때 최초로 1회 실행

roomId가 변경될 때마다 실행

Api로 부터
참여한 방 정보를 가져오는 코드

useEffect

Return()을 활용해서 컴포넌트가 unmount시 수행될 cleanup 작업을 정의할 수도 있다.

예시 : 소켓을 활용한 채팅

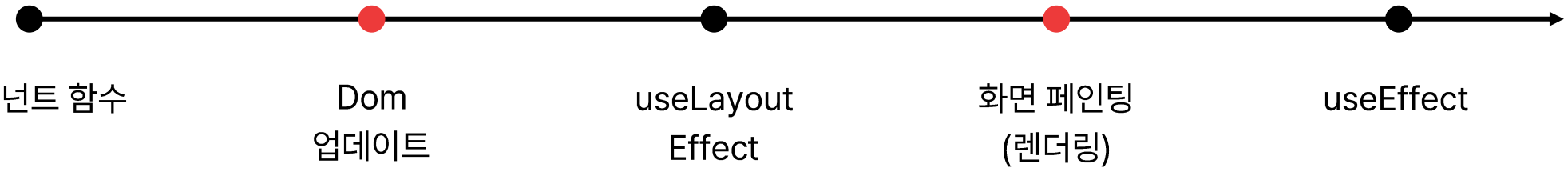
컴포넌트 Mount시 : 이벤트 리스너 추가

```
    chatSocket.on('room_update', handleRoomUpdate);
    chatSocket.on('user_join', handleUserJoin);
    chatSocket.on('user_leave', handleUserLeave); // 리스너 추가
    chatSocket.on('message_deleted', handleMessageDeleted);
    chatSocket.on('user_typing_start', handleTypingStart);
    chatSocket.on('user_typing_stop', handleTypingStop);
    return () => {
        chatSocket.off('room_update', handleRoomUpdate);
        chatSocket.off('user_join', handleUserJoin);
        chatSocket.off('user_leave', handleUserLeave); // 리스너 제거
        chatSocket.off('message_deleted', handleMessageDeleted);
        chatSocket.off('user_typing_start', handleTypingStart);
        chatSocket.off('user_typing_stop', handleTypingStop);
    };
}, [roomId, myId, members]); // members 의존성 추가
```

컴포넌트 UnMount시 : 이벤트 리스너 제거

useLayoutEffect

useEffect와 비슷한 기능을 하지만 실행 시점이 다르다.



컴포넌트 렌더링 전에 미리 필요한 값들을 계산 가능 -> 지연 없는 렌더링

렌더링 이전 실행 : Dom tree를 조작해 UI를 임의로 변경 가능

useRef

Rendering과 무관하게 상태를 관리하기 위한 Hook .

useState : 상태를 변경 -> re-rendering이 수반됨

렌더링 없이 값만 변경되어야 하는 상태를 관리하기 위해 useRef 사용

(functional 에서 포인터 같은 개념)

useRef를 사용하는 이유

Timer 객체 등 값이 자주 변해야 하는

Race Condition 관리

Race Condition 관리

```
export default function ChatInput({ roomId, myId, onMessageSent }: ChatInput
    const [input, setInput] = useState('');
    const [pending, setPending] = useState<null | { id: number; url
    const [isUploading, setIsUploading] = useState(false);
    const imageInputRef = useRef<HTMLInputElement>(null);
    const fileInputRef = useRef<HTMLInputElement>(null);
    const typingTimeoutRef = useRef<NodeJS.Timeout | null>(null);
    const hasSentTypingStartRef = useRef(false);

    // 소켓으로 타이핑 이벤트 전송
    const sendTypingEvent = (type: 'typing_start' | 'typing_stop') =
        if (chatSocket.isConnected?.() && myId) { // myId가 있을 때만
            chatSocket.send?.({ type, user_id: myId });
        }
    };
```

입력중 ... 기능 구현을 위한 현재 타이핑 상태 저장

```
// 입력값이 변경될 때마다 타이핑 상태 관리
useEffect(() => {
    if (input.trim().length > 0 && !hasSentTypingStartRef.current) {
        sendTypingEvent('typing_start');
        hasSentTypingStartRef.current = true;
    }

    if (typingTimeoutRef.current) {
        clearTimeout(typingTimeoutRef.current);
    }
```

(useRef 이름).currrent를 통해 값을 변경하거나 값에 접근할 수 있다.

useMemo / useCallback

연산 최적화를 위한 Memoization을 위해 useMemo, useCallback 등과 같은 훅을 활용할 수 있다.

useMemo : 값을 메모이제이션 하기 위한 Hook

```
function ExpensiveComponent({ num }) {
  const [count, setCount] = useState(0);

  const double = useMemo(() => {
    console.log("무거운 계산 실행...");
    return num * 2;
  }, [num]);
```

의존성 배열의 값이 바뀔 때 마다 값을 다시 계산

useCallback : 함수를 메모이제이션 하기 위한 Hook

```
function Parent() {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    console.log("clicked");
  }, []); // 의존성 없음 → 항상 같은 함수 유지
```

의존성 배열의 값이 바뀔 때 마다 함수를 다시 생성

useContext

전역 상태 관리를 위한 Hook

Ex) 로그인페이지를 제외한 어플리케이션 전체에서 user의 정보 (State)는 일정하게 관리되어야 한다.
-> 모든 페이지에서 fetch 해서 사용하는 것은 비효율적이다.

```jsx
import { createContext } from "react";

const ThemeContext = createContext("light");
export default ThemeContext;
```


```jsx
import ThemeContext from "./ThemeContext";

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

Homework

React 공식 문서 <기본 개념> 읽으며 복습하기 (Optional)
https://react.dev/learn

Hw4_api 의 구상하기 + **Hook을 이용하여 view 까지 구현하기**
명언 목록 api를 통해 자신만의 어플리케이션을 만들어봅시다.

The End

감사합니다