

# React.js

## Core Concepts

### Deep Dive

모던 React 개발의 필수 구성 요소 리뷰

Functional Components

# 1. UI의 기초 (Foundations of UI)

## Components

**컴포넌트:** React 애플리케이션의 기본 구성 요소입니다. 최신 React는 UI 요소를 반환하는 자바스크립트 함수인 `Functional Components` 를 주로 사용합니다.

## JSX

**자바스크립트 XML:** JavaScript 내에서 HTML과 유사한 코드를 허용하는 문법 확장입니다. 이는 컴파일 시 `React.createElement()` 호출로 변환됩니다.

## Curly Braces {}

**증괄호 표현식:** JSX 마크업 내에 동적인 JavaScript 표현식, 로직 또는 변수(Variable)를 직접 삽입하는 메커니즘입니다.

## Fragments

**프래그먼트:** DOM에 불필요한 노드(예: `<div>`)를 추가하지 않고 여러 자식 요소를 그룹화하는 패턴 `<>...</>` 입니다.

## 2. 데이터 흐름 및 계층 (Data Flow)

### Props (Properties)

부모(Parent) 컴포넌트에서 자식(Child) 컴포넌트로 전달되는 읽기 전용 데이터입니다.

React의 단방향 데이터 흐름(Unidirectional Data Flow)을 보장합니다.

### Children

컴포넌트의 여는 태그와 닫는 태그 사이에 요소를 직접 전달할 수 있게 하는 특수한 prop인 `props.children`입니다.

이를 통해 컴포넌트 합성(Composition)이 가능해집니다.

## 3. 동적 렌더링 (Dynamic Rendering)

### Keys (키)

리스트(List)를 렌더링할 때 필요한 고유 문자열 속성입니다. React가 변경, 추가 또는 제거된 항목을 식별하여 효율적인 재조정(Reconciliation)과 성능 최적화를 수행하도록 돕습니다.

### Rendering (렌더링)

DOM을 업데이트하는 과정입니다. 상태나 props에 따라 다른 UI를 표시하는 조건부 렌더링(Conditional Rendering)(예: `&&` 또는 삼항 연산자 사용)을 포함합니다.

## 4. 상호작용 (Interactivity)

### Event Handling (이벤트 처리)

React의 합성 이벤트(Synthetic Event) 시스템(예: `onClick`, `onSubmit`)입니다. 브라우저 간의 차이를 정규화하여 모든 환경에서 일관된 동작을 보장합니다.

### State (상태)

컴포넌트 내부에서 관리되며 시간이 지남에 따라 변할 수 있는 데이터입니다. 상태가 변경되면(via `useState`), React는 새로운 데이터를 반영하기 위해 컴포넌트를 다시 렌더링(Re-render)합니다.

## 5. 폼 패턴 (Form Patterns)

### Controlled Components (제어 컴포넌트)

기존 HTML `<input>`은 자체 상태를 가집니다. React 제어 컴포넌트에서는 변경 가능한 상태를 React의 `state`에 유지하고 `setState()`를 통해서만 업데이트합니다.

```
input value={name}  
onChange={(e) => setName(e.target.value)}
```

**Single Source of Truth (단일 진실 공급원):** React 컴포넌트가 입력 값의 유일한 출처가 되어, 실시간 유효성 검사(Validation) 및 포맷팅을 용이하게 합니다.

# 6. 흑 시스템 (The Hook System)

## Hooks (훅)

함수형 컴포넌트(Functional Component)에서 React state와 생명주기 기능(Lifecycle features)을 "연동(Hook into)"할 수 있게 해주는 함수입니다.  
(예: `useState`, `useReducer`).

## Effects (이펙트)

`useEffect`로 관리됩니다. 데이터 가져오기(Fetching), 구독(Subscription), 수동 DOM 조작과 같은 **사이드 이펙트(Side Effects)**를 처리합니다.

```
useEffect(() => {
  const loadData = async () => {
    const data = await fetchData();
    setData(data);
  };

  loadData(); // 내부에서 정의하고 호출
}, []);
```

# useEffect

useEffect`는 React Hook 중 하나로, **컴포넌트의 사이드 이펙트(Side Effect)를 처리**하기 위해 사용합니다.

**사이드 이펙트란** 컴포넌트의 렌더링과 직접적인 관련이 없는 작업들

- **API 호출**: 서버에서 데이터 가져오기
- **DOM 조작**: 직접 DOM 요소에 접근하여 조작
- **구독(Subscription)**: 이벤트 리스너 등록/해제
- **타이머 설정**: `setTimeout`, `setInterval` 등
- **localStorage 접근**: 브라우저 저장소 읽기/쓰기
- **외부 라이브러리 연동**: 차트, 지도 등

```
useEffect(() => {
  async function loadUser() {
    try {
      setLoading(true);
      setError(null);
      const userData = await getCurrentUser();
      setUser(userData);
      setFormData({ name: userData.name });
    } catch (err) {
      setError(err.message || '사용자 정보 fetch 실패');
      console.error('사용자 정보 로드 오류:', err);
    } finally {
      setLoading(false);
    }
    loadUser();
  }, []); // 빈 의존성 배열 → 마운트 시 1회만 실행
```

## 7. 직접 접근 및 전역 데이터

### Refs (참조)

**useRef** : 리렌더링을 유발하지 않고 DOM 노드에 직접 접근하거나 값을 유지하는 방법입니다.

### Context (컨텍스트)

일일이 Props를 내려주지 않고도 컴포넌트 트리 전체에 데이터를 제공할 수 있는 방법입니다. 테마나 인증 정보 같은 전역 데이터를 다룰 때 Prop Drilling 문제를 해결합니다.

# useRef

## 1. 리렌더링 없이 값 저장

- `current` 값이 변경되어도 컴포넌트가 리렌더링되지 않음
- 타이머 ID, 이전 값, 스크롤 위치 등 저장에 적합

## 2. DOM 요소 직접 접근

- `ref` 속성을 통해 DOM 요소에 접근
- `input.focus()`, `div.scrollTop` 등 DOM API 사용 가능

## 3. 컴포넌트 생명주기 동안 동일한 참조 유지

- 리렌더링 되어도 동일한 객체 참조 유지
- 이전 값 추적에 유용

```
// useRef로 input 요소 참조 생성
const nameInputRef = useRef(null);

// 수정 모드로 전환될 때 input에 포커스
useEffect(() => {
  if (isEditing && nameInputRef.current) {
    // DOM 요소에 직접 접근하여 포커스
    nameInputRef.current.focus();
  }
}, [isEditing]);
return (
  <form onSubmit={handleUpdate}>
    <input
      ref={nameInputRef} // ref 속성으로 연결
      id="name"
      type="text"
      name="name"
      value={formData.name}
      onChange={handleChange}
    />
  </form>
);
```

## 8. 코드 품질 및 무결성

### Purity (순수성)

React는 컴포넌트가 props와 state에 대해 순수 함수(Pure Function)라고 가정합니다. 동일한 입력은 항상 동일한 JSX 출력을 반환해야 하며, 이는 UI를 예측 가능하고 테스트하기 쉽게 만듭니다.

### Strict Mode

개발 모드 전용 도구(`<React.StrictMode>`)로, 컴포넌트를 의도적으로 두 번 호출하여 사이드 이펙트를 감지하고 안전하지 않은 생명주기나 deprecated API 사용을 식별합니다.

# 실습

1. 사용자 등록 (추가)
2. 사용자 목록 조회
3. 사용자 상세 조회
4. 사용자 정보 수정

1. API 서버 정보 확인
  - Base URL: `http://13.220.93.143:8080`
  - 인증: JWT Bearer Token 필요
2. API 함수 확인 (`src/utils/api.js`)
  - `getAllUsers()` - 모든 사용자 목록 조회
  - `createUser(userData)` - 사용자 추가
  - `updateUserAdmin(userId, userData)` - 사용자 정보 수정
3. 필요한 Hook 이해
  - **React 기본 Hook:**
    - `useState`: 상태 관리
    - `useEffect`: 사이드 이펙트 처리
  - **React Router Hook** (React Router 라이브러리에서 제공):
    - `useParams`: URL 파라미터 추출
    - `useSearchParams`: 쿼리 파라미터 관리
    - `useNavigate`: 프로그래밍 방식 네비게이션

## 실습 : UserDetail.jsx 리펙토링

1. 파일 길이: 430줄로 너무 길어 가독성 저하
2. 복잡도: 등록/수정/조회 로직이 하나의 파일에 혼재
3. 재사용성: 폼 로직과 UI가 결합되어 재사용 어려움
4. 테스트: 단위 테스트 작성이 어려움
5. 유지보수: 특정 기능 수정 시 전체 파일을 확인해야 함

# 컴포넌트 분리

## UserDetailHeader

← 목록으로 사용자 상세 정보 설정

사용자 ID  
1

이름  
회원가입\_테스트\_사용자

이메일  
yeon97@gmail.com

역할  
관리자

상태  
활성

가입일  
2025년 12월 10일

수정일  
2025년 12월 14일

← 목록으로 사용자 상세 정보

이름 \*  
회원가입\_테스트\_사용자

이메일 \*  
yeon97@gmail.com

역할  
관리자

상태  
활성

저장 취소

← 목록으로 새 사용자 등록

이름 \*

이메일 \*

비밀번호 \*  
 최소 6자 이상 입력해주세요.

역할  
일반사용자

상태  
활성

사용자 등록 취소

UserInfo

Userform

## 로직 분리

### hooks/useUserDetail.js - 사용자 데이터 로드

```
export function useUserDetail(userId, isCreateMode) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(!isCreateMode);
  const [error, setError] = useState(null);
  useEffect(() => {
    if (!isCreateMode && userId) {
      loadUser();
    }
  }, [userId, isCreateMode]);
  async function loadUser() {
    // 사용자 데이터 로드 로직
  }
  return { user, loading, error, refetch: loadUser };
}
```

## 로직 분리 hooks/useUserForm.js - 폼 상태 및 제출 로직

```
export function useForm(initialData, isCreateMode, onSubmitSuccess) {  
  const [formData, setFormData] = useState(initialData);  
  const [loading, setLoading] = useState(false);  
  const [error, setError] = useState(null);  
  const [success, setSuccess] = useState(false);  
  
  function handleChange(e) { // 폼 입력 처리 }  
  async function handleSubmit(e) { // 폼 제출 로직 }  
  function resetForm() { // 폼 초기화 }  
  return { formData, loading, error, success, handleChange, handleSubmit, resetForm, };  
}
```