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 {}

중괄호 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.children입니다.

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 state에 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. 훅 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)

useEffect로 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

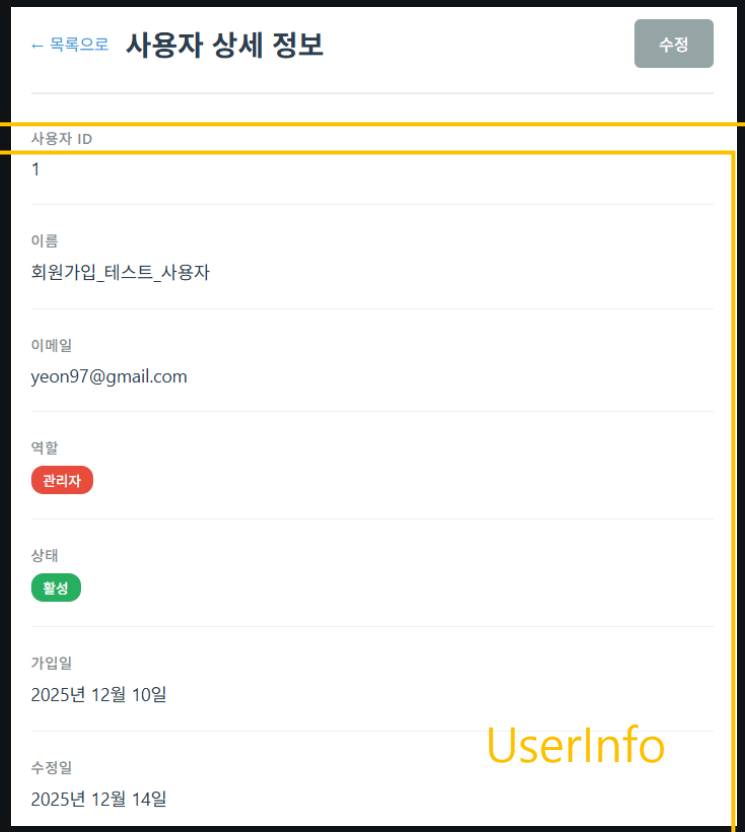
1. API 서버 정보 확인
- Base URL: `http://13.220.93.143:8080`
- 인증: JWT Bearer Token 필요
2. API 함수 확인 (`src/utils/api.js`)
- `getAllUsers()` - 모든 사용자 목록 조회
- `createUser(userData)` - 사용자 추가
- `updateUserAdmin(userId, userData)` - 사용자 정보 수정
3. 필요한 Hook 이해
- **React 기본 Hook**:
- `useState`: 상태 관리
- `useEffect`: 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



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 useUserForm(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, };
}
```

9. 고급 패턴 (Advanced Patterns)

Portals (포탈)

부모 컴포넌트의 DOM 계층 구조 외부에 존재하는 DOM 노드로 자식을 렌더링합니다.
모달(Modals) 및 툴팁(Tooltips) 구현에 필수적입니다.

Suspense (서스펜스)

코드나 데이터 로딩이 완료될 때까지 선언적으로 "대기(wait)"할 수 있게 하며, 대기하는 동안 스피너 같은 대체 UI(Fallback UI)를 보여줍니다.

Error Boundaries

하위 컴포넌트 트리 어디에서든 발생하는 자바스크립트 에러를 포착하고 기록하며, 깨진 컴포넌트 트리 대신 폴백 UI(Fallback UI)를 표시하는 컴포넌트입니다.

1. Portals: DOM 계층 탈출

문제점 :

모달(Modal)이나 툴팁(Tooltip)은 시각적으로 화면 최상단에 위치해야 함. 하지만 부모 컴포넌트에 overflow: hidden 이나 낮은 z-index가 적용되어 있으면, UI가 잘리거나 뒤에 가려지는 문제가 발생.

Stacking Context Hell: CSS만으로는 해결하기 어려운 계층 구조 문제

createPortal API를 사용하여 컴포넌트의 렌더링 위치를 물리적으로 이동시킵니다.

- **첫 번째 인자:** 렌더링할 자식 (JSX)
- **두 번째 인자:** 실제 렌더링 될 DOM 노드 (보통 document.body)

```
import { createPortal } from 'react-dom';

const ModalPortal = ({ children }) => {
// 렌더링 위치를 body로 지정
   return createPortal(
       <div className="modal">
          {children}
       </div>, document.body );
};
```

JavaScript(Vanilla JS) 모달(Modal) 구현

모달은 기술적으로 "현재 페이지 위에 떠 있는 새로운 레이어(Layer)"입니다.

이를 구현하기 위해 2가지 핵심 요소가 필요합니다.

1.Backdrop (배경): 화면 전체를 덮는 반투명한 검은색 막 (뒤쪽 컨텐츠 클릭 방지).

2.Container (컨텐츠): 실제 내용이 담긴 하얀색 박스 (화면 중앙 정렬).

Step 1: HTML 구조

모달은 보통 <body> 태그의 가장 아래쪽에 배치하여 다른 요소들의 스타일 (특히 position: relative나 overflow: hidden) 영향을 받지 않도록 합니다.

```
<!-- 모달을 여는 버튼 -->
<button id="open-btn">모달 열기</button>

<!-- 모달 전체 영역 (기본적으로 숨김 상태) -->
<div id="modal-wrapper" class="modal-overlay hidden">
  <div class="modal-content">
    <h3>안녕하세요!</h3>
    <p>바닐라 자바스크립트 모달입니다.</p>
    <button id="close-btn">닫기</button>
  </div>
</div>
```

Step 2: CSS (핵심 스타일링)

모달 구현의 90%는 CSS position 속성에 있습니다.

- position: fixed: 스크롤을 내려도 화면에 고정되게 합니다.
- z-index: 다른 요소들보다 위에 뜨게 합니다.
- display: flex: 내용물을 화면 정중앙에 쉽게 배치합니다.

```css
/* 화면 전체를 덮는 배경 */
.modal-overlay {
  position: fixed;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0, 0, 0, 0.5); /* 반투명 검정 */

  display: flex;              /* Flexbox로 중앙 정렬 */
  justify-content: center;    /* 가로 중앙 */
  align-items: center;        /* 세로 중앙 */
  z-index: 1000;              /* 가장 위에 배치 */
}

/* 실제 모달 박스 */
.modal-content {
  background: white;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 2px 10px rgba(0, 0, 0, 0.3);
}

/* 숨김 클래스 */
.hidden {
  display: none;
}
```

Step 3: JavaScript (동작 제어)

DOM 요소를 선택하고, 클래스(hidden)를 넣었다 뺐다(toggle) 하면서 보여주고 숨깁니다.

```
// 1. 요소 선택
const openBtn = document.getElementById("open-btn");
const closeBtn = document.getElementById("close-btn");
const modal = document.getElementById("modal-wrapper");

// 2. 모달 열기 함수
function openModal() {
  modal.classList.remove("hidden"); // display: none 제거
}

// 3. 모달 닫기 함수
function closeModal() {
  modal.classList.add("hidden"); // display: none 추가
}

// 4. 이벤트 연결
openBtn.addEventListener("click", openModal);
closeBtn.addEventListener("click", closeModal);
```

```
// 5. (중요 UX) 배경 클릭 시 닫기
// 모달 박스(modal-content)가 아닌 검은 배경(overlay)을 눌렀을 때만 닫혀야 함
modal.addEventListener("click", (e) => {
  if (e.target === modal) {
    closeModal();
  }
});

// 6. (중요 UX) ESC 키 누르면 닫기
window.addEventListener("keydown", (e) => {
  if (e.key === "Escape") {
    closeModal();
  }
});
```

2. Suspense: 우아한 로딩 처리

Declarative Loading

기존에는 컴포넌트 내부에서 if (isLoading) return <Spinner />와 같은 분기 처리를 일일이 해야 했습니다.

Suspense를 사용하면, 데이터가 준비되지 않았을 때 React가 알아서 렌더링을 "중단(Suspend)"하고 상위에서 지정한 **Fallback UI**(스켈레톤 등)를 보여줍니다.

```
// AS-IS: 컴포넌트마다 로딩 상태 관리 (명령형)
if (loading) return <Spinner />;

if (error) return <ErrorMessage />;

// TO-BE: Suspense로 관심사 분리 (선언형)
<Suspense fallback={<SkeletonUI />}> <UserProfile /> </Suspense>
```

자식 컴포넌트도 Suspense를 지원하는 방식(React Query, Next.js 등)으로 데이터를 가져와야 한다

3. Error Boundaries: 에러 격리

자바스크립트 에러 하나가 전체 앱을 "흰 화면(White Screen)"으로 만들어서는 안 됩니다.

Error Boundary는 하위 컴포넌트 트리에서 발생하는 에러를 포착하여, 깨진 UI 대신 **대체 UI(Fallback)**를 보여주는 React 컴포넌트입니다.

* try-catch 문의 컴포넌트 버전이라고 생각하면 쉽습니다.

```
class ErrorBoundary extends Component {
        state = { hasError: false };
        // 1. 에러 감지 시 상태 업데이트
       static getDerivedStateFromError(error) {
          return { hasError: true };
        }
        // 2. 에러 로깅 (Sentry 등)
        componentDidCatch(error, info) {
                logErrorToMyService(error, info);
         } render() {
                if (this.state.hasError) {
                        return <FallbackUI />;
                 }
                return this.props.children; } }
```

클래스기반 컴포넌트

```
import React, { Component } from 'react';

class CounterClass extends Component {
  constructor(props) {
    super(props);
        this.state = { count: 0 };  // State는 무조건 객체({}) 형태여야 함
  }

  increment = () => {
    // this.setState를 통해서만 변경 가능
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        {/* this 키워드 필수 사용 */}
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>증가</button>
      </div>
    );
  }
}
```

클래스형 컴포넌트 (Legacy)

- React.Component를 **상속(extends)** 받아야 합니다.
- 반드시 render() 메서드가 있어야 하며, 그 안에서 JSX를 리턴합니다.
- this.state라는 **객체** 하나에 모든 상태를 몰아서 관리합니다.
- constructor(생성자)에서 초기값을 설정합니다

생명주기(Lifecycle) vs useEffect

클래스형 컴포넌트에서는 컴포넌트의 인생(생성 -> 업데이트 -> 사망)을 관리하는 특정 메서드들이 있었습니다.

함수형에서는 이를 useEffect 하나로 통합했습니다.

시점	클래스형 컴포넌트 (메서드)	함수형 컴포넌트 (Hooks)
마운트 (생성)	`componentDidMount()`	`useEffect(() => { ... }, [])`
업데이트 (변화)	`componentDidUpdate()`	`useEffect(() => { ... }, [value])`
언마운트 (제거)	`componentWillUnmount()`	`useEffect(() => { return () => { ... } }, ...)`

클래스형 컴포넌트의 단점 (왜 함수형으로 넘어왔나?)

1. **this 키워드의 혼란:** 자바스크립트의 this는 상황에 따라 다르게 동작하여 초급자에게 큰 진입 장벽이었습니다. (이벤트 핸들러 바인딩 문제 등)
2. **재사용의 어려움:** 로직을 재사용하려면 HOC(Higher-Order Component)나 Render Props 같은 복잡한 패턴을 써야 했습니다. -> 함수형은 **Custom Hooks**로 쉽게 해결했습니다.
3. **코드 길이:** 같은 기능을 구현해도 클래스형이 훨씬 코드가 깁니다 (Boilerplate).
4. **Minification:** 빌드 시 코드 압축이 함수형보다 덜 효율적입니다.

react-error-boundary

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

function Fallback({ error, resetErrorBoundary }) {
  return (
    <div role="alert">
      <p>에러가 발생했어요:</p>
      <pre>{error.message}</pre>
      <button onClick={resetErrorBoundary}>다시 시도</button>
    </div>
  );
}

const App = () => (
  <ErrorBoundary FallbackComponent={Fallback}>
    <MyFunctionalComponent />
  </ErrorBoundary>
);
```