6차시 스크립트

State Management & 프로젝트 구조화

학습 목표

- React Router로 페이지 이동 기능 구현 방법을 이해한다.
- 프로젝트 구조를 명확하게 설계한다.
- 상태 관리 도구의 개념(Context API, Redux)을 이해한다.
- 실제 레시피 앱에 간단한 페이지 라우팅을 구현한다.

1️⃣ React Router: 페이지 이동의 기초

📌 개념 설명

- SPA(Single Page Application)는 한 HTML 안에서 다양한 컴포넌트를 보여주는 방식
- React Router는 페이지처럼 보이도록 컴포넌트 전환을 처리
- React는 기본적으로 SPA → URL 변화가 있어도 새 페이지를 요청하지 않음
- 실제로는 여러 페이지처럼 보여야 함 (ex: Home, About, Favorites 등)\

react-router-dom은 **브라우저의 History API(pushState)**를 활용

→ 주소창은 바뀌되, 실제 HTTP 요청은 발생하지 않음

→ 바뀐 URL을 감지해 특정 컴포넌트를 렌더링함

[BrowserRouter] → [Route Matching] → [Element Rendering] 순으로 작동.

1. `BrowserRouter`

- HTML5 history API를 사용하여 URL 변경 감지

2. `Routes`, `Route`

- `Route`들은 실제 매칭되는 URL이 있을 때만 렌더링
- 동적 파라미터 (`:id`)는 `useParams()`로 접근 가능

3. 코드 분할 (Code Splitting)

React Router는 `lazy()`와 `Suspense`로 **route-based 코드 분할**을 쉽게 할 수 있음

```
tsx
CopyEdit
import { lazy, Suspense } from 'react';
const Settings = lazy(() => import('./pages/Settings'));

<Route path="/settings" element={
  <Suspense fallback={<Loading />}>
    <Settings />
  </Suspense>
} />
```

왜 코드 분할이 필요한가?

- SPA는 모든 JS 코드를 한 번에 다운받으면 **초기 로딩이 느림**
- 페이지에 진입하지도 않았는데 무거운 컴포넌트까지 미리 받으면 낭비

React에서의 해법: `React.lazy()` + `Suspense`

- `React.lazy()`는 동적으로 컴포넌트를 import하여, **해당 라우트가 호출되기 전까지 로딩하지 않음**
- `Suspense`는 로딩 중일 때 보여줄 UI를 지정

4. 중첩 라우팅 구조

```
<Route path="/dashboard" element={<DashboardLayout />}>
  <Route index element={<DashboardHome />} />
  <Route path="stats" element={<DashboardStats />} />
</Route>
```

- `/dashboard` 접근 시 `DashboardLayout`이 기본으로 렌더링됨
- 내부 `<Outlet />`을 통해 하위 컴포넌트를 전환함

정리

- `BrowserRouter` → URL 변경 감지
- `Routes` → 가장 먼저 매칭되는 `Route` 탐색
- `Route` → 매칭되면 해당 컴포넌트 렌더링

📦 설치 및 설정

```
npm install react-router-dom
```

```
// src/App.js
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Home from "./pages/Home";
import About from "./pages/About";

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

🔶 `<Link />` 를 통한 이동

```
import { Link } from 'react-router-dom';

<Link to="/about">About</Link>
```

📁 디렉토리 구조 개선

```
src/
├── components/      # 재사용 가능한 UI 컴포넌트
│   └── RecipeCard.js
├── pages/           # 페이지별 컴포넌트
│   ├── Home.js
│   └── About.js
├── store/           # 상태 관리 (zustand)
│   └── useRecipeStore.js
├── api/             # API 요청 정리
│   └── recipeApi.js
├── App.js
└── index.js
```

3️⃣ 상태 관리: 왜 필요한가?

🔍 문제 상황

- props drilling: 부모 → 자식 → 자식 → 자식...
- 여러 컴포넌트가 동일한 데이터를 참조하거나 수정해야 함
- API 요청 결과를 앱 전반에서 공유해야 함
- 컴포넌트 간 데이터 공유를 돕는 시스템
- **데이터 흐름을 예측 가능하게 하고, 중복 제어를 제거**함

4️⃣ 상태 관리 도구 소개

📘 1. Context API

- 리액트에 기본 내장, Provider로 전역 공유
- **장점**: 간단하고 의존성 없음
- **단점**: 상태가 커질수록 비효율, 최적화 어려움
- `createContext`와 `Provider`로 감싸고, `useContext`로 접근

```
// context/ThemeContext.js
```

```
const ThemeContext = createContext();

export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('dark');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
```

🧪 사용 예시

```
// App.js
<ThemeProvider>
  <App />
</ThemeProvider>

// components/Header.js
const { theme } = useContext(ThemeContext);
```

📘 2. Redux

- 액션(action) → 리듀서(reducer) → 스토어(store)
- **장점**: 정형화된 구조, 디버깅 용이 (Devtools), 미들웨어 활용 가능
- **단점**: 보일러플레이트 많고 초보자에겐 러닝커브 있음

Redux 개념도

```
UI → dispatch(action) → reducer → store → state 업데이트 → UI 리렌더링
```

요소	설명
Action	상태 변경을 설명하는 객체. `{ type: "INCREMENT" }`

Reducer	현재 상태와 액션을 받아 새 상태를 리턴하는 함수
Store	앱의 전역 상태 트리 보관소. 모든 상태는 여기에 저장
Dispatch	액션을 발생시키는 함수. `dispatch({ type: "ADD" })`

Redux 개념 코드 흐름

```
// store.js
import { createStore } from 'redux';

const initialState = { count: 0 };

function reducer(state = initialState, action) {
  switch (action.type) {
    case 'INCREMENT': return { count: state.count + 1 };
    default: return state;
  }
}

const store = createStore(reducer);
```

```
// index.js
import { Provider } from 'react-redux';
<Provider store={store}>
  <App />
</Provider>
```

```
// Component
const count = useSelector((state) => state.count);
const dispatch = useDispatch();
```

- **단방향 데이터 흐름 (unidirectional)**
- 코드 분리와 정형화가 강점이나, 소규모 프로젝트에는 과함

5️⃣ 실습: Zustand로 상태 관리 도입

🔷 Zustand 소개

- Redux보다 훨씬 간단한 API
- hook 기반으로 전역 상태 사용
- 빠르고, 가볍고, 직관적

📦 설치

```
bash
CopyEdit
npm install zustand
```

✅ store/useRecipeStore.js

```
import { create } from 'zustand';

const useRecipeStore = create((set) => ({
  recipes: [],
  setRecipes: (data) => set({ recipes: data }),
}));

export default useRecipeStore;
```

✅ 사용 예시

```
// pages/Home.js
import useRecipeStore from '../store/useRecipeStore';
import { useEffect } from 'react';
```

```jsx
import axios from 'axios';

export default function Home() {
  const { recipes, setRecipes } = useRecipeStore();

  useEffect(() => {
    axios.get('https://www.themealdb.com/api/json/v1/1/search.php?s=')
      .then((res) => {
        setRecipes(res.data.meals);
      });
  }, []);

  return (
    <div>
      <h1>레시피 목록</h1>
      {recipes.map((r) => (
        <div key={r.idMeal}>{r.strMeal}</div>
      ))}
    </div>
  );
}
```

📁 실습 폴더 구조 예시

```
src/
├── pages/
│   ├── Home.js
│   └── About.js
├── store/
│   └── useRecipeStore.js
```

📚 정리

- React Router는 SPA를 페이지처럼 보이게 해주는 도구

- 상태 관리는 복잡한 앱에서 필수 → 상태를 전역으로, 예측 가능하게
- Context API는 간단한 전역 상태 공유엔 적합하지만 최적화 한계
- Redux는 정형화된 대규모 구조에 적합
- Zustand는 **현대적이고 가벼운 상태 관리의 좋은 선택**

🔥 "상태(state)"란 대체 뭐냐?

- 컴포넌트의 내부 데이터: UI와 사용자 인터랙션을 연결하는 중심
- 예시: `const [count, setCount] = useState(0);`

이렇게 생긴 앱이 점점 커지면?

- 다양한 컴포넌트에서 같은 데이터를 참조하고 수정해야 함
- 데이터가 바뀌면 해당되는 UI가 자동으로 리렌더링 되어야 함

✅ 로컬 상태 vs 전역 상태

- **로컬 상태 (Local State):** 컴포넌트 내부 `useState`, `useReducer`

  → 예: 입력 폼, 모달 열림 여부 등
- **전역 상태 (Global State):** 여러 컴포넌트가 공유하는 상태

  → 예: 로그인 정보, 테마, 장바구니, 언어 설정 등

💥 문제: 상태 공유의 폭발

⛓️ 1. Prop drilling의 고통

```
jsx
CopyEdit
<App>
  <Header>
    <CartIcon itemCount={props.itemCount} />
  </Header>
</App>
```

- 깊은 컴포넌트 트리로 상태를 계속 전달해야 함
- 중간에 사용하는 컴포넌트는 그 데이터를 **쓸모도 없지만** 계속 받음

🧠 상태 관리 도구가 해결하려는 진짜 문제

문제	상태 관리 도구의 해결책
Prop drilling	전역 상태 store로 직접 접근
리렌더링 범위	필요한 곳만 리렌더 (최적화)
구조 복잡도	명확한 흐름으로 예측 가능하게
협업 시 혼란	역할 분리, 정형화된 패턴 제공

🧭 왜 Redux는 그렇게 복잡하게 만들어졌는가?

1️⃣ 단방향 데이터 흐름의 철학

```
사용자 → Action → Store → UI 업데이트
```

- 어떤 변화든 반드시 `dispatch()`를 거쳐야 한다
- 사이드 이펙트는 middleware(thunk, saga)로 분리

2️⃣ Redux는 예측 가능성 + 디버깅을 위해 복잡함을 감수함

- 모든 상태 변화는 log로 남음 → time-travel debugging 가능
- 앱이 커질수록 이런 명시적 구조가 유리해짐

➡️ 단점: 보일러플레이트, 반복 코드, 초기 러닝커브

❌ Context API만으로는 왜 부족할까?

Context는 좋지만 **두 가지 문제**가 있다:

1. 모든 Consumer가 리렌더링됨

```
<SomeProvider>
  <ComponentA /> ← context 사용하는 곳
  <ComponentB /> ← context 안쓰는데도 같이 리렌더됨
</SomeProvider>
```

- 상태가 바뀔 때마다 모든 하위 컴포넌트가 리렌더링
- useMemo, React.memo로 막을 수 있지만 유지보수가 힘들어짐

2. Provider 중첩이 지옥

```
<AuthProvider>
  <ThemeProvider>
    <LanguageProvider>
      <UserProvider>
        <App />
```

- 코드 가독성 ↓, 디버깅 어려움 ↑

➡️ Context는 **소규모, 단순한 전역 상태**에 적합함

🪶 Zustand는 왜 더 경량인가?

1️⃣ Zustand는 상태를 custom hook처럼 관리

```
const useStore = create((set) => ({
  count: 0,
  inc: () => set((state) => ({ count: state.count + 1 })),
}));
```

- 컴포넌트에서 `useStore((state) => state.count)`처럼 부분 접근 가능
- 구독한 값만 바뀌면 해당 컴포넌트만 리렌더링됨

2️⃣ Redux와 비교되는 설계 철학

항목	Redux	Zustand
설계 철학	**예측 가능한 복잡한 앱**	**단순하고 빠른 글로벌 상태**
상태 접근	dispatch → reducer	hook 호출만으로 접근
분기 처리	switch-case reducer	그냥 set 함수로

미들웨어	thunk/saga 별도 필요	비동기 바로 가능
사용 난이도	높음 (보일러플레이트 있음)	매우 쉬움 (1 파일로 끝남)