React
실습 프로젝트

AI 지출 관리 애플리케이션

React 19.2.0 Vite 7.2.4 2024 Project

개발 환경: Vite와 pnpm

Vite

- **Instant Server Start:** 번들링 없는 즉시 시작
- **Lightning Fast HMR:** 수정 사항 즉시 반영
- **ES Modules:** 브라우저 네이티브 모듈 활용

pnpm

- **Fast Installation:** npm 대비 3배 빠른 속도
- **Disk Efficient:** 중복 없는 심볼릭 링크 구조
- **Strict Deps:** 유령 의존성(Phantom Deps) 방지

Dev Server Startup

< 300 ms (vs Webpack 30s+)

Disk Space Usage

- 80 % (Content Addressable Store)

프로젝트 개요

React의 핵심 개념을 실무 중심으로 학습하기 위해 설계된 **AI 지출 관리 애플리케이션**입니다.

> 사용자 관리, 지출 내역 추적, 영수증 AI 분석 등의 기능을 통해 **Hooks, Context API, React Router, 비동기 처리** 등 React의 주요 개념을 실제로 구현하고 경험할 수 있습니다.

Tech Specification

Project Name	react-expense-app
React	v19.2.0
Vite	v7.2.4
React Router	v6.21.1
Spring Boot	v20.x (LTS)

구현된 주요 기능

인증 시스템

로그인, 회원가입, 로그아웃 기능 구현.
Context API를 활용한 전역 인증 상태 관리 및 Protected Route 적용.

사용자 관리

관리자 권한의 사용자 목록 조회, 등록, 상세 조회, 수정 등 완전한 CRUD 기능 구현.

지출 내역 관리

개인별 지출 목록 조회 및 상세 정보 확인.
영수증 이미지 업로드 및 미리보기 기능 제공.

OpenAI API 연동

OpenAI SDK를 활용한 영수증 이미지 분석 및 데이터 추출.
프롬프트 엔지니어링을 통한 구조화된 응답 처리.

기타 유틸리티

반응형 UI 설계로 모바일/데스크탑 지원.
URL 쿼리 파라미터를 활용한 검색 및 필터링 기능.
로컬 상태를 활용한 간단한 Task 관리.

React Hooks 기초

함수형 컴포넌트에서 상태(State)와 생명주기(Lifecycle) 기능을 사용할 수 있게 해주는 핵심 도구들입니다

useState

Role

컴포넌트의 로컬 상태를 관리하며, 상태 변경 시 컴포넌트 리렌더링을 트리거합니다.

Project Usage

모든 컴포넌트
폼 입력값, 로딩 상태(loading), 에러 메시지, 모달 표시 여부 관리

useEffect

Role

데이터 로딩, 구독, DOM 수정 등 컴포넌트의 사이드 이펙트(Side Effect)를 처리합니다.

Project Usage

UserList, AuthContext
마운트 시 API 데이터 호출, localStorage에서 사용자 세션 복원

useContext

Role

컴포넌트 트리 전체에 데이터를 제공하여 Prop Drilling 문제를 해결합니다.

Project Usage

AuthContext
로그인 사용자 정보와 인증 토큰을 전역 상태로 관리하여 공유

성능 최적화 Hooks

useMemo

계산 비용이 큰 **값(Value)**을 메모이제이션합니다.
의존성 배열의 값이 변경될 때만 함수를 재실행하여 불필요한 연산을 방지합니다.

EXAMPLE

```
const memoizedValue = useMemo(() => {
    returncomputeExpensiveValue(a, b); }, [a, b]);
```

Project Case

UserDetail 컴포넌트에서 사용자 정보 필터링 및 가공 시 사용하여 렌더링 성능 확보

useCallback

함수(Function) 자체를 메모이제이션합니다.
자식 컴포넌트에 props로 함수를 전달할 때, 참조 동등성을 유지하여 불필요한 리렌더링을 막습니다.

EXAMPLE

```
const memoizedCallback = useCallback(() => { doSomething(a, b); }, [a, b]);
```

Project Case

AuthContext의 login, logout 함수를 메모이제이션하여 Context Consumer들의 재렌더링 최소화

Custom Hooks 설계

로직 재사용과
관심사의 분리

- 로직 재사용성 증가
- 컴포넌트 복잡도 감소
- 테스트 용이성 향상
- 비즈니스 로직 캡슐화

useUserDetail

Data Fetching

사용자 상세 정보를 로드하고 관리하는 로직을 캡슐화하여 컴포넌트에서 API 호출 코드를 분리함.

Returns { user, loading, error }

Usage UserDetail.jsx

useUserForm

Form Logic

폼 입력값 관리, 유효성 검사, 제출 처리 로직을 통합하여 반복되는 폼 핸들링 코드를 제거함.

Returns { formState, handleChange, handleSubmit }

Usage UserForm.jsx

Context API 전역 상태 관리

AuthContext 구조

Context 생성: createContext() 로 전역 저장소 생성

Provider: 앱 최상위에서 상태와 함수를 하위 컴포넌트에 주입

Custom Hook: useAuth() 로 간편한 접근 인터페이스 제공

관리되는 상태 (State)

User: 현재 로그인한 사용자 정보 객체

Token: API 인증을 위한 JWT 토큰

IsAuthenticated: 로그인 여부 (Boolean)

주요 기능

Prop Drilling 해결: 깊은 컴포넌트 트리까지 props 전달 불필요

상태 지속성: localStorage와 연동하여 새로고침 시 상태 복원

AuthContext.jsx

```
export const AuthContext = createContext();
export function AuthProvider({ children }) {
    const [user, setUser] = useState(null);  // 로그인 함수 메모이제이션
    const login = useCallback(async (email, pw) => {
            const data = await api.post('/login', { email, pw });
            setUser(data.user);
            localStorage.setItem('token', data.token); }, []);
            const value = useMemo(() => ({
                    user, login, isAuthenticated: !!user }
            ),  [user, login]);
    return (
            <AuthContext.Providervalue={value}>
                    {children}
            </AuthContext.Provider> ); }
```

React Router 기초

BrowserRouter

HTML5 History API를 사용하여 UI와 URL을 동기화하는 최상위 라우터 컴포넌트입니다.

Routes & Route

현재 URL과 일치하는 경로(path)를 찾아 해당 컴포넌트(element)를 렌더링합니다.

중첩 라우팅 (Nested Routes)

부모 라우트 내부에 자식 라우트를 정의하여 레이아웃을 공유하고 계층적인 UI를 구성합니다.

App.jsx

```
<BrowserRouter>
  <Routes>
    // 공통 레이아웃 적용
    <Route path="/" element={<Layout />}>
      <Route index element={<Home />} />
      // 보호된 라우트 (인증 필요)
      <Route element={<ProtectedRoute />}>
        <Route path="users" element={<UserList />} />
      </Route>
    </Route>
  </Routes>
</BrowserRouter>
```

동적 라우팅과 쿼리 파라미터

/users/123 ?mode=edit

useParams() useSearchParams()

useParams

URL 경로의 동적 세그먼트(Dynamic Segment)를 객체 형태로 추출합니다.

```
// Route path="/users/:id"
  const { id } = useParams();
  useEffect(() => { // ID가 변경될 때마다 데이터 로드
      fetchUser(id); }, [id]);
```

useSearchParams

URL의 쿼리 스트링을 읽거나 수정할 수 있는 인터페이스를 제공합니다. (useState와 유사)

```
const [searchParams, setSearchParams] = useSearchParams(); const mode =
searchParams.get('mode'); constswitchToEdit = () => { setSearchParams({ mode:
'edit' }); };
```

프로그래매틱 네비게이션

useNavigate

사용자 이벤트(클릭, 폼 제출) 처리 후 **명령형(Imperative)**으로 페이지를 이동시킬 때 사용합니다.

```
const navigate = useNavigate();

const handleLogin = async () =>
{ await login(email, password);
// 로그인 성공 후 대시보드로 이동
(
// replace: true로 뒤로가기 방지
navigate('/dashboard', { replace: true }); };
```

useLocation

현재 URL의 경로(pathname), 쿼리 파라미터(search), 상태(state) 등 **위치 정보**를 반환합니다.

```
const location = useLocation(); useEffect(() => {
// 페이지 이동 시마다 실행 (예: GA 추적)
console.log('Current Path:', location.pathname);
// 이전 페이지에서 전달된 상태 확인
if(location.state?.from) { showToast('Redirected from protected
page'); } }, [location]);
```

컴포넌트 설계 패턴

컴포넌트 분리

Page: 데이터 로딩 및 레이아웃 (UserList)

UI: 재사용 가능한 순수 뷰 (UserForm)

Layout: 공통 구조 및 네비게이션

데이터 흐름 (Props)

Parent → Child: 데이터 및 상태 전달

Child → Parent: 콜백 함수로 이벤트 전달

Context: 전역 상태 공유 (Auth)

Conditional Rendering

조건부 렌더링

```
return (
<div>
 {isLoading ?<Spinner />:<Content />}
 {error &&<ErrorMessage />}
</div>
 );
```

List Optimization

리스트 렌더링 최적화

```
{users.map(user => (
<UserItem
key={user.id} /* 고유 키 필수 */
user={user}
onSelect={handleSelect}
/>
 ))}
```

폼 관리와 제어 컴포넌트

신뢰 가능한
단일 출처 (Source of Truth)

React State를 폼 데이터의 유일한 출처로 사용하여, 입력값과 UI를 완벽하게 동기화합니다.

- 사용자 입력 (Input)
- onChange 이벤트 발생
- setState로 상태 업데이트
- 리렌더링 및 value 반영

Login.jsx

```
functionLoginForm() { // 1. 상태 선언
const [email, setEmail] = useState(''); // 2. 이벤트 핸들러
consthandleChange = (e) => {
  setEmail(e.target.value); // 실시간 유효성 검사 가능 };
return ( <input type="email"// 3. 상태와 UI 바인딩
                value={email} onChange={handleChange} /> ); }
```

비동기 처리와 API 연동

Request Lifecycle

1. Initialization

로딩 상태를 true로 설정하고 에러를 초기화합니다.

2. Execution

비동기 API 호출을 수행하고 응답을 기다립니다 (await).

3. Handling

성공 시 데이터를 저장하고, 실패 시 에러를 포착 (catch)합니다.

4. Finalization

결과에 상관없이 로딩 상태를 false로 변경합니다.

PATTERN

```
const fetchUsers = async () => {
    setLoading(true);
    setError(null);
    try { // API 호출 및 대기
            const response = await api.get('/users');
          // 성공: 데이터 상태 업데이트.
          setUsers(response.data);
    } catch (err) { // 실패: 에러 상태 업데이트
            setError(err.message);
           console.error('Fetch failed:', err);
    } finally { // 종료: 로딩 상태 해제
       setLoading(false); } };
```

파일 업로드 처리

FileReader API

클라이언트 측에서 파일을 비동기적으로 읽어 **이미지 미리보기**(Data URL)를 생성합니다.

FormData API

파일과 텍스트 데이터를 multipart/form-data 형식으로 인코딩하여 서버로 전송합니다.

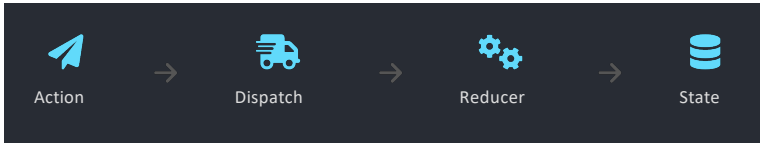
File Input → Preview → Upload

Upload.jsx

```
// 미리보기 생성
const reader = new FileReader();
reader.onloadend = () => {
setPreview(reader.result);
};
reader.readAsDataURL(selectedFile);

// FormData 생성 (이미지 파일 업로드용)
const formData = new FormData();
formData.append('image', imageFile);
```

useReducer: 복잡한 상태 관리



구분	useState	useReducer
복잡도	단순 값 (원시타입)	복잡한 객체/로직
업데이트	직접 값 설정 (setter)	액션(Action) 발행
로직 분리	컴포넌트 내부	외부 함수로 분리 가능
디버깅	어려움	액션 로그 추적 용이

```
// 1. Reducer 함수 정의 (순수 함수)
function reducer(state, action) {
    switch (action.type) {
      case 'INCREMENT':
          return { count: state.count + 1 };
      case 'DECREMENT':
         return { count: state.count - 1 };
       default:
         return state; }
 }
function Counter() {
  // 2. Hook 사용
    const [state, dispatch] = useReducer(reducer, { count: 0 });
   return (
     <> Count: {state.count} // 3. Action 디스패치
        <button onClick={() => dispatch({ type: 'INCREMENT' })}> + </button>
</>
 ); }
```