⚛️ React Redux

📌 1. Redux란?

✅ 정의

Redux는 **React 애플리케이션에서 상태(state)를 중앙 집중식으로 관리하기 위한 라이브러리**입니다.

- 단방향 데이터 흐름을 유지하며, 상태의 추적이 쉽고 디버깅이 용이함
- 상태 관리의 복잡도를 줄이고 컴포넌트 간의 props 전달을 최소화할 수 있음

✅ 주요 개념

개념	설명
Store	전역 상태가 저장되는 객체
Action	상태 변경을 알리는 객체 (type 필수)
Reducer	Action에 따라 상태를 어떻게 변경할지 정의하는 함수
Dispatch	Action을 Reducer로 전달하는 함수
useSelector	store의 상태를 가져오는 hook
useDispatch	Action을 발생시키는 hook

🔧 2. Redux 기본 세팅

✅ 설치

npm install @reduxjs/toolkit react-redux

✅ Redux Toolkit으로 store 생성

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './features/counterSlice';

const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
});

export default store;
```

✅ Provider로 앱에 store 주입

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

📦 3. Slice 생성 (Redux Toolkit 방식)

```
// features/counterSlice.js
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => { state.value += 1; },
    decrement: (state) => { state.value -= 1; },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    },
  },
```

```
});

export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

🧪 4. 컴포넌트에서 Redux 사용하기

```
// Counter.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './features/counterSlice';

function Counter() {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <h2>카운터: {count}</h2>
      <button onClick={() => dispatch(increment())}>증가</button>
      <button onClick={() => dispatch(decrement())}>감소</button>
    </div>
  );
}

export default Counter;
```

✅ `useSelector`로 상태 조회, `useDispatch`로 액션 발생

🧩 5. 비동기 처리 (createAsyncThunk)

```
// features/userSlice.js
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import axios from 'axios';

export const fetchUser = createAsyncThunk('user/fetchUser', async () => {
  const res = await axios.get('https://jsonplaceholder.typicode.com/users/1');
  return res.data;
});

const userSlice = createSlice({
  name: 'user',
  initialState: { data: null, loading: false },
```

```
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUser.pending, (state) => {
        state.loading = true;
      })
      .addCase(fetchUser.fulfilled, (state, action) => {
        state.data = action.payload;
        state.loading = false;
      });
  },
});

export default userSlice.reducer;
```

📦 6. 설계 전략

✅ Slice 파일 구조

```
src/
├── features/
│   ├── authSlice.js
│   ├── userSlice.js
│   └── counterSlice.js
```

✅ 상태 관리 설계 시 고려사항

- 글로벌 vs 로컬 상태 판단
- 복잡한 데이터 흐름은 thunk로 모듈화
- persist (redux-persist) 활용해 새로고침에도 상태 유지 가능
- RTK Query를 사용하면 API 호출과 상태 관리를 통합적으로 처리 가능

🏁 요약

✅ 핵심 정리

- Redux는 컴포넌트 간 상태 공유, 예측 가능한 상태 관리에 매우 유용
- Redux Toolkit을 활용하면 설정이 간단하고 유지보수가 쉬움
- useSelector/useDispatch 훅을 통해 함수형 컴포넌트와 쉽게 연동 가능
- 비동기 작업은 createAsyncThunk를 통해 안전하게 처리 가능