

12강

React의 상태관리 Redux 사용법)

&

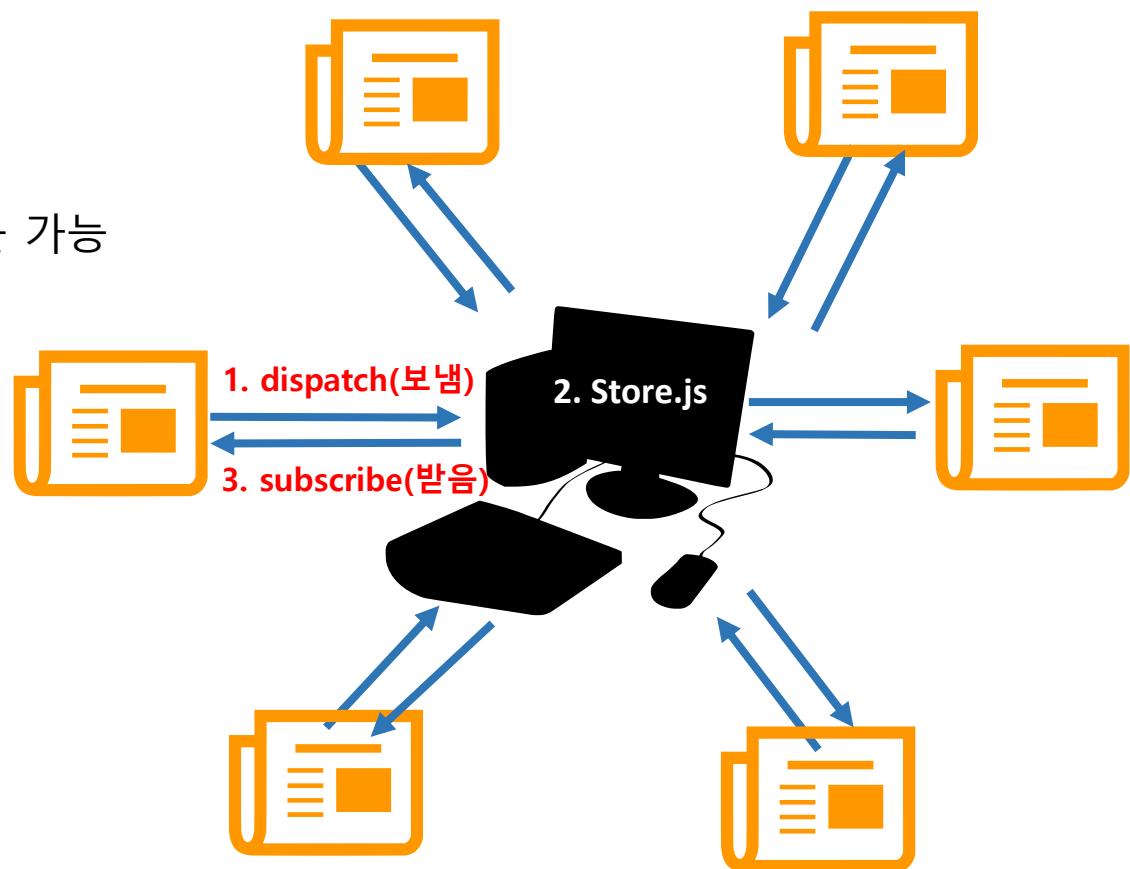
리덕스 툴킷(Redux Toolkit)





➤ Redux가 필요한 이유

- 문제상황: Props Drilling
 - 깊은 컴포넌트 트리에서 데이터 전달의 어려움
 - 중간 컴포넌트들의 불필요한 props 전달
- 해결책: Redux
 - 중앙 집중식 상태 관리
 - 어느 컴포넌트에서든 직접 접근 가능





➤ Redux란 ?

- JavaScript 애플리케이션의 전역 상태 관리 라이브러리 이다.
- “상태(State)를 한곳(Store)에 모아 관리한다.”
- Context API의 확장판 이다.

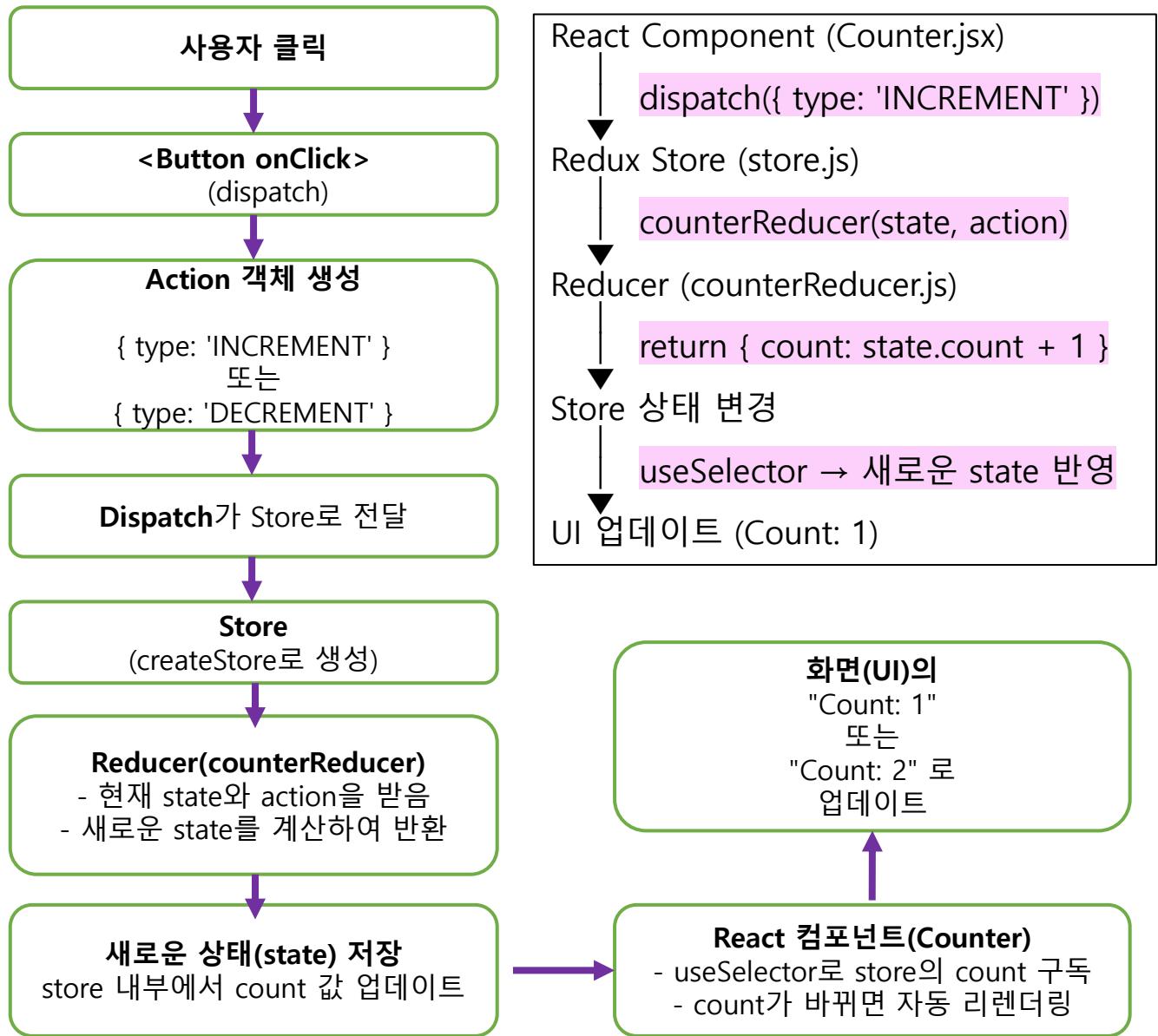
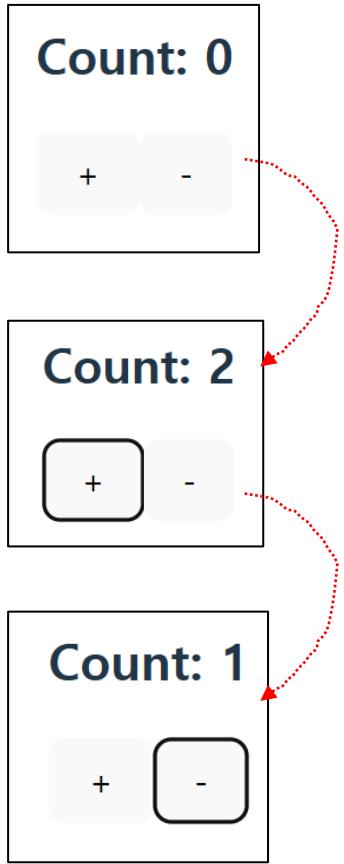
➤ Context API와 Redux 차이점

- Context API는 간단하지만 대규모 상태 관리에는 한계가 있다.
- Redux는 예측 가능한 상태 관리 구조를 제공한다.

| 항목 | Context API | Redux |
|---------|-------------|------------------------|
| 관리 단위 | 소규모 상태 | 대규모 상태 |
| 업데이트 구조 | 단순 | 엄격 (Reducer 필수) |
| 유지보수성 | 낮음 | 높음 |
| 사용 예 | 로그인, 테마 | 사용자, 장바구니, 주문, UI 상태 등 |



▶ Redux 데이터 흐름도





➤ Redux의 3가지 핵심 요소

1. Store: 상태 저장소
 - 애플리케이션의 전체 상태를 보관
 - 단 하나만 존재
2. Action: 행동
 - 상태 변경을 요청하는 객체
 - type과 payload 포함
3. Reducer: 상태 변경 함수
 - 현재 상태 + Action -> 새로운 상태
 - 순수 함수로 작성
4. Dispatch: Action을 보내는 함수

➤ Redux 설치

- `npm install redux`
- `npm install react-redux`

redux: 상태 관리 핵심

react-redux: React와 연결



➤ 액션(Action)

- 애플리케이션에서 일어나는 사건을 설명하는 객체
- **Type** 속성을 필수로 가지며, 상태 변경을 트리거(어떤 사건을 촉발시키는)하는 역할을 한다.
- 예) { type: 'INCREMENT' }, { type: 'ADD_TODO', text: 'Learn Redux' }

➤ 리듀서(Reducer)

- 액션을 처리하여 새로운 상태를 변환하는 함수
- 이전 상태와 액션 객체를 인자로 받아 새로운 상태 객체를 반환한다.
- 순수 함수여야 하며, 입력이 같으면 출력도 항상 같아야 하다.
 - 순수 함수(Pure Function)
 - 동일한 인자가 주어졌을 때 항상 동일한 결과를 반환
 - 함수의 실행이 외부 상태에 의존하지 않은
 - 외부 상태를 변경하지 않는 함수



➤ action.payload란

Redux에서 액션(action) 은 state를 변경하도록 전달되는 메시지라고 생각하면 된다.

액션은 보통 이렇게 아래와 같다

```
{  
  type: 'counter/increment', // 어떤 일을 할지 정의  
  payload: 10                // 상태를 변경할 때 필요한 데이터  
}
```

- type: 필수, 어떤 액션인지 식별자
 - payload: 선택, state를 바꾸는 데 필요한 데이터(추가 정보)
 - 즉, payload는 액션을 실행할 때 전달하는 값이다.
-
- **payload** = 액션에 담아 전달하는 추가 데이터
 - **type** = 어떤 액션인지 구분하는 식별자
 - Redux Toolkit에서 액션 생성 함수에 값을 전달하면, 자동으로 payload에 담겨서 리듀서에서 사용 가능



➤ 스토어(Store)

- 애플리케이션의 상태를 담고 있는 객체
- `createStore` 함수를 사용하여 스토어를 생성한다.
- 스토어는 3가지 메소드를 가진다.
 - `getState()` : 현재 상태 반환함.
 - `dispatch(action)` : 상태를 변경하는 액션 보내기
 - `Subscribe(listener)` : 상태가 변경될 때마다 호출되는 리스너를 등록하기

➤ .js vs .jsx의 차이

| 확장자 | 의미 | 용도 | 예시 |
|------|---------------|-------------------------------|---------------------------------|
| .js | 순수 자바스크립트 파일 | JSX 문법이 없는 로직, 설정 파일 | store.js, reducer.js, utils.js |
| .jsx | JSX 문법 포함된 파일 | React 컴포넌트 (HTML처럼 보이는 코드 있음) | App.jsx, Counter.jsx, Login.jsx |



➤ store.js 작성

```
// createStore 최소선은 앞으로는 사라질 예정이니
// 새 방식으로 바꿔 써라”는 경고 표시
// Redux 최신 버전(>= 4.2, 특히 Toolkit 통합 이후)에서는
// createStore()를 더 이상 공식적으로 권장하지 않는다.
import { createStore } from 'redux';
import counterReducer from './counterReducer';

export const store = createStore(counterReducer);
```



➤ counterReducer.js 작성

```
// 상태(state)의 초기값 설정
// - Redux에서는 반드시 초기 상태를 정의해야 함
// - 앱이 시작될 때 state가 undefined인 경우, 이 값이 기본으로 사용됨
const initialState = { count: 0 };

// 리듀서 함수 (Reducer Function)
// - "상태(state)"와 "액션(action)"을 받아서 새로운 상태를 반환하는 순수 함수
// - 기존 state를 직접 변경하지 않고, 항상 새로운 객체를 반환해야 함
export default function counterReducer(state = initialState, action) {
  switch (action.type) {
    // 카운트 증가 액션
    // { type: 'INCREMENT' } 가 전달되면 count를 +1 시킴
    case 'INCREMENT':
      return { count: state.count + 1 };

    // 카운트 감소 액션
    // { type: 'DECREMENT' } 가 전달되면 count를 -1 시킴
    case 'DECREMENT':
      return { count: state.count - 1 };

    // 그 외의 액션 (매칭되지 않는 경우)
    // state를 그대로 반환 (즉, 아무 변화 없음)
    default:
      return state;
  }
}
```



➤ Provider 적용 (main.jsx)

```
import { Provider } from 'react-redux';
import { store } from './store';
import App from './App';

<Provider store={store}>
  <App />
</Provider>
```

➤ Counter.jsx

```
import { useSelector, useDispatch } from 'react-redux';

export default function Counter() {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  return (
    <>
      <h2>Count: {count}</h2>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>-</button>
    </>
  );
}
```



➤ Redux 단점

- 액션 타입, 액션 생성자, 리듀서 등 파일과 코드가 많음
- switch-case 문 사용 → 복잡한 문법
- 상태 업데이트 시 불변성을 수동으로 처리해야 함 → 실수 가능

➤ Redux Toolkit 사용하는 이유

- Redux의 복잡성을 해결하기 위한 공식 도구
- 간단한 설정: configureStore() 하나로 스토어 설정 가능
- 더 적은 코드로 상태 관리 가능
- 현대적인 Redux 사용 패턴
- createSlice()로 액션과 리듀서를 한 번에 생성
- Immer.js 내장 → 불변성 처리 자동

➤ Redux Toolkit 설치

- `npm install @reduxjs/toolkit`



➤ Redux Toolkit의 핵심 API

| API | 설명 |
|-----------------------|--|
| configureStore() | 스토어 생성 및 미들웨어 설정을 간단하게 처리 |
| createSlice() | 상태(state), 리듀서(reducer), 액션(action)을 한 번에 생성 |
| createAsyncThunk() | 비동기 로직 처리(예: API 호출) |
| createEntityAdapter() | 리스트 상태를 효율적으로 관리 |



➤ [createSlice\(\)](#) 정리

| 속성 | 의미 | 이름 변경 가능 여부 | 값(내용) 변경 가능 여부 | 비고 |
|--------------|-----------------------|----------------|-------------------|--------------------------------|
| name | 슬라이스 이름, 액션 prefix | (예약어) | (값은 자유롭게 지정 가능) | ex) 'counter', 'todo' |
| initialState | 초기 상태값 | (예약어) | (객체 구조 자유) | ex) { value: 0 }, { list: [] } |
| reducers | 상태 변경 로직 모음 | (예약어) | (안의 함수 이름, 로직 자유) | ex) increment, decrement 등 |

① name

- 슬라이스(state)의 이름(식별자) 을 의미한다.

② initialState

- 슬라이스가 관리할 초기 데이터(상태값)를 의미한다.
- 이름은 바꾸면 안 됨 (예약 키워드), 하지만 안에 들어가는 데이터 구조는 자유롭게 정의할 수 있다.

③ reducers

- 상태(state)를 변경하는 함수들의 모음(로직 집합) 이다.
- 이름은 반드시 reducers (복수형) 이어야 한다.
- Redux Toolkit 내부에서 이 키를 인식해서 자동으로 action 생성 함수를 만든다.



➤ counterSlice.js

```
// 1단계. slice 파일 만들기 (counterSlice.js)
// Redux의 "상태 + 로직"을 정의하는 부분이다.
// 이 단계에서 state 구조, state를 변경하는 reducer 함수,
// 그리고 action 생성 함수를 정의한다.
import { createSlice } from '@reduxjs/toolkit';

const countSlice = createSlice({
  name: 'counter', //state의 이름 (store에서 접근할 때 key로 사용됨)
  initialState: { value: 0 }, //초기상태 0
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
    reset: (state) => {
      state.value = 0;
    },
  },
});
export const { increment, decrement, reset } = countSlice.actions;
console.log(countSlice.actions);
export default countSlice.reducer;
```



➤ countSlice.actions 의미 - 액션 생성자들(action creators)

- Redux가 자동으로 만들어주는 액션 생성 함수 모음이다.

```
const countSlice = createSlice({
  name: 'counter', //state의 이름 (store에서 접근할 때 key로 사용됨)
  initialState: { value: 0 }, //초기상태 0
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
    reset: (state) => {
      state.value = 0;
    },
  },
});
```

- 이렇게 작성하면, Redux Toolkit이 내부적으로 아래와 같은 “액션 생성 함수”를 자동으로 만들어 준다

```
countSlice.actions = {
  increment: () => ({ type: 'counter/increment' }),
  decrement: () => ({ type: 'counter/decrement' }),
  reset: () => ({ type: 'counter/reset' }),
}
```

- `dispatch(increment())` 라고 하면 실제로는 Redux가 이런 액션 객체를 만들어서 store에 전달한다.
`=> { type: "counter/increment" }`



➤ countSlice.reducer — “상태를 실제로 바꾸는 함수”

- Redux에서 reducer는 “현재 상태(state)”와 “액션(action)”을 받아서 새로운 상태(new state)를 반환하는 순수 함수이다.
- Redux Toolkit의 createSlice()는 reducers 안에 적어둔 로직을 자동으로 합쳐서 하나의 큰 reducer 함수로 만들어준다.

countSlice.reducer는 아래처럼 동작하는 함수를 의미한다

- 이 함수가 store 안에서 상태를 실제로 바꾸는 로직이 된다.

```
(state = { value: 0 }, action) => {
  switch (action.type) {
    case 'counter/increment':
      state.value += 1;
      break;
    case 'counter/decrement':
      state.value -= 1;
      break;
    case 'counter/reset':
      state.value = 0;
      break;
    default:
      return state;
  }
}
```



➤ 총 정리

| 코드 | 의미 |
|---|--|
| countSlice.actions | 자동 생성된 액션 함수 모음 (→ dispatch()로 호출) |
| countSlice.reducer | 상태(state)를 변경하는 실제 reducer 함수 |
| store.js에서 reducer: { counter: countSlice.reducer } | store에 reducer 등록 (state.counter로 접근 가능) |



➤ store.js

```
// 2단계. store 만들기 (store.js)
// Redux의 중앙 저장소 역할을 하는 부분이다.
// slice를 store에 등록해야 컴포넌트에서 사용할 수 있다.

import { configureStore } from '@reduxjs/toolkit';
import countReducer from './counterSlice';

export const store = configureStore({
  reducer: {
    counter: countReducer, // slice 등록 (key는 state 이름)
  },
});
```



➤ main.jsx

```
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';
import './index.css';
import App from './App.jsx';
import { Provider } from 'react-redux';
import { store } from './store.js';
// 3단계. App 전역에 store 연결하기 (main.jsx)

// React와 Redux를 연결하는 부분이다.
// <Provider>로 감싸서 하위 컴포넌트들이
// useSelector, useDispatch를 사용할 수 있게 해준다.

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



➤ Counter.jsx

```
// 4단계. 컴포넌트에서 상태 사용하기 (Counter.jsx)

// 이제 컴포넌트에서 useSelector로 상태를 읽고,
// useDispatch로 액션(increment, decrement 등)을 호출한다.
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, reset } from './counterSlice';

export default function Counter() {
  const count = useSelector((state) => state.counter.value); //상태읽기
  const dispatch = useDispatch(); //액션 실행 준비

  return (
    <div>
      <h1>카운터 : {count}</h1>
      <button type="button" onClick={() => dispatch(increment())}>+</button>
      <button type="button" onClick={() => dispatch(decrement())}>-</button>
      <button type="button" onClick={() => dispatch(reset())}>Reset</button>
    </div>
  );
}
```



➤ useSelector()의미

```
const count = useSelector((state) => state.counter.value); //상태읽기
```

- Redux의 store 안에 있는 상태(state) 를 읽어오는 역할이다.
- 즉, 전역 상태를 React 컴포넌트에서 꺼내 쓰는 코드이다.

➤ useDispatch()의미

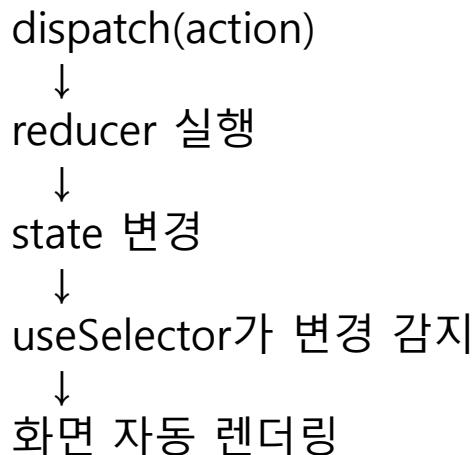
```
const dispatch = useDispatch(); //액션 실행 준비
```

- Redux store에 명령(액션)을 전달하는 함수(dispatcher)를 꺼내오는 코드이다.
 1. useDispatch()는 Redux store 내부의 dispatch() 함수를 반환한다.
(즉, store.dispatch와 동일한 역할)
 2. 우리는 이 dispatch 함수를 이용해서 액션을 store에 전달(발사) 한다



➤ 총 정리

| 코드 | 역할 |
|---|-------------------------------------|
| useSelector((state) => state.counter.value) | store의 상태를 읽어서 화면에 표시 |
| useDispatch() | store에 액션을 보낼 수 있는 함수(dispatch)를 준비 |
| dispatch(increment()) | "+1 해줘"라는 명령을 store에 보냄 |
| reducer | 이 명령을 실제로 처리해서 state를 변경 |
| React | state 변경을 감지하고 화면을 다시 그림 |





〈초기 화면〉

카운터: 0

+ - Reset

〈[+] 버튼 클릭〉

카운터: 4

+ - Reset

〈[-] 버튼 클릭〉

카운터: 2

+ - Reset

〈[Reset] 버튼 클릭〉

카운터: 0

+ - Reset



➤ Redux와 Redux Toolkit 비교

| 구분 | Redux | Redux Toolkit |
|----------|----------------------------|--------------------------------------|
| Store 생성 | <code>createStore()</code> | <code>configureStore()</code> |
| Reducer | 함수 직접 작성 | <code>createSlice()</code> 자동 생성 |
| Action | <code>type</code> 직접 작성 | 자동 생성 (<code>slice.actions</code>) |
| 코드 길이 | 길고 복잡 | 짧고 명확 |

기초 연습 문제





문제1] 다음의 조건에 만족하도록 React를 작성 하시오.

조건

- ① 장바구니 상품목록을 추가하는 기능이 구현되도록 작성하시오.
- ② 아래 오브젝트 배열을 이용해 작성하시오.

```
const sampleProducts = [  
  { id: 1, name: '사과', price: 1000 },  
  { id: 2, name: '바나나', price: 1500 },  
  { id: 3, name: '오렌지', price: 1200 },  
];
```

- ① 다음 페이지의 <출력 결과>를 확인하여 작성하시오.
- ② 버튼 클릭 시 텍스트 표시 여부 변경
- ③ src 폴더안에 -> cartEx 폴더 생성 -> Cart.jsx 파일을 생성하여 작성하시오.
- ④ src 폴더안에 -> cartEx 폴더 생성 -> cartSlice.js 파일을 생성하여 작성하시오.
- ⑤ src 폴더안에 -> store.js에 cart를 등록하여 작성하시오.
- ⑥ src 폴더안에 -> App.jsx에 작성한 Cart.jsx 를 import하여 실행 하시오.



<출력 결과>

장바구니

상품 목록

- 사과 - 1000원 장바구니 추가
- 바나나 - 1500원 장바구니 추가
- 오렌지 - 1200원 장바구니 추가

장바구니

장바구니 비우기

<상품 추가>

장바구니

상품 목록

- 사과 - 1000원 장바구니 추가
- 바나나 - 1500원 장바구니 추가
- 오렌지 - 1200원 장바구니 추가

장바구니

- 사과 - 1000원 삭제
- 오렌지 - 1200원 삭제

장바구니 비우기

<상품 삭제>

장바구니

상품 목록

- 사과 - 1000원 장바구니 추가
- 바나나 - 1500원 장바구니 추가
- 오렌지 - 1200원 장바구니 추가

장바구니

- 사과 - 1000원 삭제

장바구니 비우기