



# 10장 상태 관리

## 10.1 상태 관리

### 1. 상태 **State**

상태 : 렌더링에 영향을 줄 수 있는 동적인 데이터 값

리액트 공식 문서 : 렌더링 결과에 영향을 주는 정보를 담은 순수 JS 객체

- 리액트 내부 API만을 사용하여 상태를 관리할 수 있지만, 성능 문제, 상태의 복잡성으로 인해 외부 상태 관리 라이브러리 **Redux, MobX, Recoil** 를 주로 활용한다.
- 리액트 앱 내의 상태 : 지역 상태, 전역 상태, 서버 상태

### 지역 상태 **Local State**

컴포넌트 내부에서 사용되는 상태

ex) 체크박스의 체크 여부, 폼의 입력값

- **useState** 훅을 가장 많이 사용한다.
- 때에 따라 **useReducer** 훅을 사용하기도 한다.



#### ▼ **useReducer**

🌟 Redux가 아니라 React 내장 훅이다.

: **미니 Redux** Redux의 핵심 패턴인 reducer를 컴포넌트 레벨에서 사용할 수 있게 해주는 React 훅

### 전역 상태 **Global State**

앱 전체에서 공유하는 상태

ex) 전역 상태 관리 도구 **redux, zustand**

- 상태가 변경되면 컴포넌트들이 업데이트된다.

- **Prop drilling** 문제를 피하고자 **지역 상태**를 해당 컴포넌트들 사이의 전역 상태로 공유할 수도 있다.

Prop drilling : props를 통해 데이터를 전달하는 과정에서 중간 컴포넌트는 해당 데이터가 필요하지 X음에도 자식 컴포넌트에 전달하기 위해 props를 전달해야 하는 과정

## 서버 상태 **Server State**

사용자 정보, 글 목록 등 외부 서버에 저장해야 하는 상태들

- UI 상태와 결합하여 관리하게 되며, 로딩 여부나 에러 상태 등을 포함한다.
- 서버 상태는 지역 상태 or 전역 상태와 동일한 방법으로 관리된다. 최근에는 외부 라이브러리 **react-query** **SWR** 사용하여 관리한다.

## 2. 상태를 관리하기 위한 가이드

➤ 상태가 없는 Stateless 컴포넌트를 활용하는 게 좋다.

### ✓ 값을 상태로 정의할 때 2가지 사항

- 시간이 지나도 변하지 않는다면 상태가 아니다.
- 파생된 값은 상태가 아니다.

### ➤ 시간이 지나도 변하지 X는다면 상태가 아니다

시간이 지나도 변하지 X는 값이라면, **객체 참조 동일성**을 유지하는 방법을 고려해본다.

컴포넌트가 마운트될 때만 스토어 객체 인스턴스를 생성하고,  
컴포넌트가 언마운트될 때까지  
해당 참조가 변하지 않는다고 가정해보자.



## ▼ 참조가 변하는 상황 / 유지되는 상황

### 참조가 변하는 주요 상황

1. 상위 컴포넌트로부터 새로운 props를 전달받을 때
2. 컴포넌트 내부에서 setState를 호출할 때
3. 부모 컴포넌트가 리렌더링될 때

### 참조가 유지되는 상황

- ~~useRef, useMemo, useCallback~~

1. 외부 라이브러리 인스턴스 관리
2. 폼 상태 관리
3. 캐시/메모리 저장소
4. 애니메이션 컨트롤러

## ✓ 메모이제이션 useMemo

컴포넌트가 마운트될 때만 객체 인스턴스를 생성하고, 이후 렌더링에서는 이전 인스턴스를 재활용한다.(렌더링될 때마다 동일한 객체 참조 유지)

! 리액트 공식 문서 : 객체 참조 동일성 유지를 위해 useMemo를 사용하는 것은 권장되지 X  
는다. 오로지 **성능 향상**을 위한 용도로만 사용할 것.

🌟 useMemo 없이도 올바르게 동작하도록 작성하고, 나중에 성능 개선을 위해 useMemo를 추가하자.



## 동작 방법



### 객체 참조 동일성 유지 방법

- useState의 초기값만 지정하는 방법
- useRef를 사용하는 방법 (권장)

- **useState의 초깃값만 지정하는 방법**(~~useState 사용 X~~)

! 의미론적으로 좋은 방법은 아니다.

```
useState(new Store()) // 렌더링마다 생성
useState(() => new Store()) // 지연 초기화 방식
```

▼ 함수 컴포넌트 예시

```
// 매 렌더링마다 createStore()가 실행됨
// const [store] = useState(createStore());

// 초기 렌더링 시에만 콜백이 실행됨
const [store] = useState(() => createStore());

// createStore 함수의 예시
const createStore = () => {
  // 무거운 초기화 작업
  const heavyCalculation = someExpensiveOperation();

  return {
    data: heavyCalculation,
    items: [],
    addItem: (item) => items.push(item)
  };
}
```

**지연 초기화 방식** : 초깃값을 계산하는 콜백을 지정하는 방식

**?** **useState를 쓸 때 값이 바뀌면 렌더링 되어서 참조가 변하는데 이게 어떻게 방법이 수 있을까?**

초깃값만 지정하는 방법은 `setState`를 사용하지 않는 것이 핵심이다.

▼ 예시


```
// Store 타입 정의
type Store = {
  data: string[];
  addData: (item: string) => void;
  removeData: (index: number) => void;
};

function Component() {
  // 초깃값만 설정하고 setState는 사용하지 않음
  const [store] = useState<Store>(() => ({
    data: [],
    addData: (item: string) => store.data.push(item)
    removeData: (index: number) => store.data.splice
  }));

  // 이렇게 직접 data 배열을 수정
  const handleClick = () => {
    store.addData("새 항목");
    // 여기서 forceUpdate나 다른 state를 통해 리렌더링을 트
  };

  return <div>{store.data.length}</div>;
}
```

- **useRef** 를 사용하는 방법

 동일한 객체 참조를 유지하려는 목적으로 사용하기에 적합한 훅

```
// useRef의 인자로 직접 넣으면, 렌더링마다 인스턴스 생성
import {useRef} from 'react';

const store = useRef<Store>(null);
```

```
if (!store.current) {
  store.current = new Store();
}
```

#### ▼ 함수 컴포넌트

```
// 잘못된 방식
const store = useRef<Store>(createStore());

// 올바른 방식
const store = useRef<Store | null>(null);

if (!store.current) {
  store.current = createStore();
}
```

## ➤ 파생된 값은 상태가 아니다

파생된 값 : 부모에게서 전달받을 수 있는 props, 기존 상태에서 계산될 수 있는 값

**SSOT** (Single Source Of Truth) : 어떠한 데이터도 단 하나의 출처에서 생성하고 수정해야 한다는 원칙을 의미하는 방법론

- SSOT를 지키는 예시

두 컴포넌트에서 동일한 데이터를 상태로 갖고 있을 경우 ~~(자식 컴포넌트에서 해당 상태를 바꿔야 하는데 동기화 되어야 한다.)~~

! 자식 컴포넌트에서 props로 받은 값을 다시 useState **상태** 로 선언하면 독자적으로 관리되므로 안된다.

✗ useEffect로 동기화 : useEffect를 사용한 동기화 작업은 리액트 외부 데이터 **로컬스토리지** 와 동기화할 때만 사용해야 한다. 내부에 존재하는 데이터를 상태와 동기화하면 안된다!

✓ 데이터를 동기화하면 안되고, 단일한 출처에서 데이터를 사용하도록 변경해줘야 한다.



**상태 끌어올리기** : 상위 컴포넌트에서 상태를 관리하도록 해주는 기법

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

```

type UserEmailProps = {
  email: string;
  setEmail: React.Dispatch<React.SetStateAction<string>>;
};

const UserEmail: React.VFC<UserEmailProps> = ({ email, setEmail }) => {
  const onChangeEmail = (event: React.ChangeEvent<HTMLInputElement>) => {
    setEmail(event.target.value);
  };
  return (
    <div>
      <input type='text' value={email} onChange={onChangeEmail} />
    </div>
  );
};

```

아이템 목록과 선택된 아이템 목록을 가지고 있는 코드

:

**아이템 목록** 이 변경될 때마다 **선택된 아이템 목록** 을 가져오기 위해 `useEffect` 로 동기화 작업을 한다.

```

import {useState, useEffect} from 'react';

const [items, setItems] = useState<Item[]>([]);
const [selectedItems, setSelectedItems] = useState<Item[]>([]);

useEffect(() => {
  setSelectedItems(items.filter((item) => item.isSelected));
}, [items]);


```

! **items** 와 **selectedItems** 가 동기화되지 않을 수 있다!

! 게다가 **setSelectedItems** 를 사용하여 **items**에서 가져온 데이터가 아닌 임의의 데이터셋을 설정하는 것도 구조적으로 가능하며 오류가 발생할 수 있다.

! 성능 측면 : **items** 와 **selectedItems** 2가지 상태를 유지하면서 `useEffect`로 동기화하는 과정을 거친다. → **selectedItems** 값을 얻기 위해 **렌더링이 2번 발생**한다.

1. **items**의 값이 바뀌며 렌더링 발생 **items** 상태 변경

 상태로 정의하지 않고 계산된 값을 JS 변수로 담는다.

items가 변경될 때마다 컴포넌트가 새로 렌더링하고, 매번 렌더링될 때마다 selectedItems를 다시 계산하게 된다!

```
import {useState} from 'react'

const [items, setItems] = useS
const selectedItems = items.f
```

2. items의 값이 변경됨을 감지하고, selectedItems 값을 변경하며 리렌더링 발생 selectedItems 상태 변경

✗ JS 변수에 계산 결과를 담으면 리렌더링 횟수는 줄지만, 매번 렌더링될 때마다 계산을 수행한다.

✓ useMemo : items가 변경될 때만 계산을 수행하고 결과를 메모이제이션하여 성능을 개선하자!

```
import {useState, useMemo} from

const [items, setItems] = useS
const selectedItems = useMemo
```

## useState vs useReducer, 어떤 것을 사용해야 할까

✓ useState 대신 useReducer 사용을 권장하는 경우

- 다수의 하위 필드를 포함하고 있는 복잡한 상태 로직을 다룰 때
- 다음 상태가 이전 상태에 의존적일 때

배달의민족 리뷰 리스트를 필터링하여 보여주기 위한 쿼리를 상태로 지정해야 하는 상황

- 쿼리의 하위 필드 : 검색날짜범위, 리뷰점수, 키워드, 페이지네이션(페이지, 사이즈)...

```
// 날짜 범위 기준 - 오늘, 1주일, 1개월
type DateRangePreset = 'TODAY' | 'LAST_WEEK' | 'LAST_MONTH';

type ReviewRatingString = '1' | '2' | '3' | '4' | '5';

interface ReviewFilter {
  // 리뷰 날짜 필터링
  startDate: Date;
  endDate: Date;
```



```

dateRangePreset: Nullable<DateRangePreset>;
// 키워드 필터링
keywords: string[];
// 리뷰 점수 필터링
ratings: ReviewRatingString[];
// ... 이외 기타 필터링 옵션
}
// Review List Query State
interface State {
  filter: ReviewFilter;
  page: string;
  size: number;
}

```

### ! useState

useState로 다르면 상태를 업데이트할 때마다 잠재적인 오류 가능성이 증가한다.

- 의도치 않게 다른 필드가 수정될 수도 있다.
- 특정한 업데이트 규칙이 있다면 useState만으로는 한계가 있다.  
ex) '사이즈 필드를 업데이트할 때는 페이지 필드를 0으로 설정해야 한다.'

### 👩 useReducer

useReducer : '무엇을 변경할지' **action** 와 '어떻게 변경할지' **reducer** 를 분리하여, **dispatch** 를 통해 어떤 작업을 할지를 **action** 으로 넘기고 **Reducer** 함수 내에서 상태를 업데이트한다.

```

import React, { useReducer } from 'react';

// Action 정의
// 업데이트를 위한 정보를 가지고 있다. (dispatch의 인자)
type Action =
  | { payload: ReviewFilter; type: 'filter'; }
  | { payload: number; type: 'navigate'; }
  | { payload: number; type: 'resize'; };

// Reducer 정의
const reducer: React.Reducer<State, Action> = (state, action) =>
  switch (action.type) {

```

```

    case 'filter':
      return {
        filter: action.payload,
        page: 0,
        size: state.size,
      };
    case 'navigate':
      return {
        filter: state.filter,
        page: action.payload,
        size: state.size,
      };
    case 'resize':
      return {
        filter: state.filter,
        page: 0,
        size: action.payload,
      };
    default:
      return state;
  }
};

// useReducer 사용
const [state, dispatch] = useReducer(reducer, getDefaultState())

// dispatch 예시
dispatch({ payload: filter, type: 'filter' });
dispatch({ payload: page, type: 'navigate' });
dispatch({ payload: size, type: 'resize' });

```

boolean 상태를 토글하는 액션만 사용하는 경우  
 → useState 대신  
 useReducer 를 사용하곤 한다.

```
import { useReducer } from 'react';
```

```
//Before
const [fold, setFold] = useState(true);

const toggleFold = () => {
  setFold((prev) => !prev);
};

// After
const [fold, toggleFold] = useReducer((v) => !v, true);
// dispatch
// reducer 함수
// 초기값 true
// action 객체가 필요 없으면 생략 가능.
// toggleFold(); -> dispatch 호출과 같음!!
```

### 3. 전역 상태관리와 상태관리 라이브러리

상태는 사용하는 곳과 최대한 가까워야 하며 사용 범위를 제한해야만 한다.

✓ 전역 상태로 사용 방법

- Context API + `useState` or `useReducer`
- 외부 상태관리 라이브러리 `Redux`, `MobX`, `Recoil`

#### Context API

전역적으로 공유해야 하는 데이터를 컨텍스트로 제공하고 해당 컴포넌트를 구독한 컴포넌트에서만 데이터를 읽을 수 있게 된다.

- Context API는 엄밀하게 말해 전역 상태 관리 솔루션이라기보다 여러 컴포넌트 간에 값을 공유하는 솔루션이다.

`TabGroup` 컴포넌트와 `Tab` 컴포넌트들에게 `type`이라는 prop을 전달한 경우



`TabGroup` 컴포넌트에만 이 prop을 전달하고, `Tab` 컴포넌트들의 구현 내에서도 사용할 수 있게 하고 싶다.

```
// 현재 구현된 것 - TabGroup 컴포넌트뿐 아니라 모든 Tab 컴포넌트에도 type
<TabGroup type='sub'>
  <Tab name='탭 레이블 1' type='sub'>
    <div>123</div>
  </Tab>
  <Tab name='탭 레이블 2' type='sub'>
    <div>123</div>
  </Tab>
</TabGroup>

// 원하는 것 - TabGroup 컴포넌트에만 전달
<TabGroup type='sub'>
  <Tab name='탭 레이블 1'>
    <div>123</div>
  </Tab>
  <Tab name='탭 레이블 2'>
    <div>123</div>
  </Tab>
</TabGroup>
```

상위 컴포넌트 `TabGroup` 구현부에 Context Provider를 넣어주고, 하위 컴포넌트 `Tab` 에서 해당 컨텍스트를 구독하여 데이터를 읽어온다.

```
import {FC} from 'react';

//(추가 예시)////////////////////////////////////
// Context 생성
type TabGroupContextType = {
  type: 'tab' | 'button';
  // ... 다른 props들
};

const TabGroupContext = createContext<TabGroupContextType | null>
////////////////////////////////////

const TabGroup: FC<TabGroupProps> = (props) => {
  const { type = 'tab', ...otherProps } = useTabGroupState(props)
  /* ... 로직 생략 */
  return (
```

```

    <TabGroupContext.Provider value={{ ...otherProps, type }}>
      {/* ... */}
    </TabGroupContext.Provider>
  );
};

const Tab: FC<TabProps> = ({ children, name }) => {
  const { type, ...otherProps } = useTabGroupContext();
  return <>{/* ... */}</>;
};

```



### Context API 팁!

**유틸리티 함수**를 정의하여 더 간단한 코드로 컨텍스트와 훅을 생성할 수 있다.

createContext라는 **유틸리티 함수**를 정의해서 자주 사용되는 **프로바이더**와 **해당 컨텍스트를 사용하는 훅**을 한번에 간편하게 생성하여 같은 코드를 반복해서 작성하지 않아도 된다.

```

import React from 'react';

// 컨텍스트 값을 가져오는 함수의 타입을 정의
type Consumer<C> = () => C;

// 컨텍스트가 가질 기본적인 형태를 정의
export interface ContextInterface<S> {
  state: S;
}

// S : 우리가 관리할 상태의 타입
// C : 컨텍스트의 타입
export function createContext<S, C = ContextInterface<S>>(): React.Context<C> {
  // 실제 컨텍스트 생성
  const context = React.createContext<Nullable<C>>(null); // type: Context<C>

  // Provider 컴포넌트 생성
  // : 자식 컴포넌트들에게 컨텍스트 값을 제공하는 컴포넌트
  const Provider = ({ children }: { children: React.ReactNode }) => {
    return (
      <ContextInterface.Provider
        value={context.state}
        children={children}
      />
    );
  };

  return { context, Provider };
}

```

```

const Provider: React.FC<C> = ({ children, ...otherProps }) =>
  // children: <ChildComponent />
  // otherProps: { state: { currentTab: 'home', isOpen: true
  return (
    <context.Provider value= {otherProps as C}>{children}</cor
  );
};

// useContext hook 생성
// : 컨텍스트 값을 가져오는 커스텀 hook
const useContext: Consumer<C> = () => {
  const _context = React.useContext(context);
  if (!_context) {
    throw new Error(ErrorMessage.NOT_FOUND_CONTEXT);
  }
  return _context;
};

return [Provider, useContext];
}

// Example
interface StateInterface {}
const [context, useContext] = createContext<StateInterface>();

```

#### ▼ 유틸리티 함수 사용 예시

```

// 상태 타입 정의
interface TabState {
  currentTab: string;
  isOpen: boolean;
}

// Provider와 Hook 생성
const [TabProvider, useTab] = createContext<TabState>();

// 사용
function TabComponent() {
  return (

```

```

    <TabProvider state={{ currentTab: 'home', isOpen: true }}
      <ChildComponent />
    </TabProvider>
  );
}

function ChildComponent() {
  const { state } = useTab(); // { currentTab: 'home', isOpen: true }
  return <div>{state.currentTab}</div>;
}

```



변수에 `_` 접두사를 붙이는 이유

1. 이름 충돌 방지
2. 임시 변수나 내부 변수임을 표시

참고: 사용하지 않는 매개변수를 나타낼 때도 `_`를 자주 사용한다.

### ✓ 전역상태관리 솔루션 := ContextAPI + `useState`, `useReducer`

Context API는 전역 상태 관리 솔루션이라기보다 여러 컴포넌트 간에 값을 공유하는 솔루션이다.

하지만! 지역상태관리 API `useState`, `useReducer` 와 결합하여 여러 컴포넌트 사이에서 상태를 공유하기 위한 방법으로 사용된다.

```

import { useReducer } from 'react';

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <StateProvider.Provider value={{ state, dispatch }}>
      <ComponentA />
      <ComponentB />
    </StateProvider.Provider>
  );
}

```

! 대규모 애플리케이션이나 성능이 중요한 애플리케이션에서 Context API를 사용하여 전역 상태를 관리하는 것은 권장되지 X한다.

→ 프로바이더의 props로 주입된 값이나 참조가 변경될 때마다 해당 컨텍스트를 구독하고 있는 **모든 컴포넌트**가 리렌더링되기 때문이다.

(Zustand는 필요한 상태만 선택적 구독 가능, ContextAPI는 Provider의 값이 하나라도 변경될 때 모든 구독 컴포넌트 리렌더링)

## 10.2 상태 관리 라이브러리

### 1. MobX

객체 지향 프로그래밍, 반응형 프로그래밍 패러다임의 영향을 받은 라이브러리

- **장점** 객체 지향 스타일로 코드를 작성하는 데 익숙하다면 추천
- **단점** 데이터 추적이 어렵기 때문에 트러블슈팅의 어려움이 있다.

### 2. Redux

함수형 프로그래밍의 영향을 받은 라이브러리

- **장점** 특정 UI 프레임워크에 종속되지 않아 독립적으로 라이브러리를 사용할 수 있다.
- **장점** 상태 변경 추적이 최적화 → 특정 상황에서 발생한 애플리케이션 문제의 원인 파악 용이
- **단점** 단순한 상태 설정에도 많은 보일러플레이트 필요

| 보일러플레이트 : 최소한의 변경으로 여러 곳에서 재사용되는 코드

- **단점** 사용 난도가 높다.

### 3. Recoil

상태를 저장할 수 있는 **Atom** 과 해당 상태를 변형할 수 있는 순수 함수 **Selector** 를 통해 상태를 관리하는 라이브러리

- **장점** Redux에 비해 보일러플레이트가 적고, 난이도가 쉽다.
- **단점** 실험적인 상태(충분한 검증X)



## 4. Zustand

Flux 패턴을 사용하며 많은 보일러플레이트를 가지지 않는 혹은 기반의 편리한 API 모듈을 제공한다.

**클로저**를 활용하여 스토어 내부 상태를 관리함으로써 특정 라이브러리에 종속되지 않는 특성이 있다.

상태, 상태를 변경하는 액션을 정의하고 반환된 혹은 어느 컴포넌트에서나 임포트하여 원하는 대로 사용한다.

### ▼ 코드

- set 함수 : Zustand store의 상태를 업데이트하는 함수

```
// 2가지 방식
// 1. 직접 새 상태 객체 전달
set({ bears: 0 })

// 2. 이전 상태를 기반으로 새 상태 계산
set((state) => ({ bears: state.bears + 1 })))
```

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

const useBearStore = create((set) => ({
  bears: 0,
  increasePopulation: () => set((state) => ({ bears: state.bears + 1 })),
  removeAllBears: () => set({ bears: 0 }),
}));

//
function BearCounter() {
  const bears = useBearStore((state) => state.bears);

  return <h1>{bears} around here ...</h1>;
}

//
function Controls() {
  // 1 상태값이나 함수를 가져오는 방법1
  const increasePopulation = useBearStore((state) => state.increasePopulation);
}
```

```
// 즉, increasePopulation 함수만 구독하므로 해당 상태가 변경될 때만

// 2 상태값이나 함수를 가져오는 방법2
const { increasePopulation } = useBearStore(); // 모두 가져와
// store의 어떤 상태가 변경되더라도 컴포넌트가 리렌더링 ex) bears 값

return <button onClick={increasePopulation}>Plus</button>;
}
```

## 우형 이야기: 상태관리라이브러리

### ▼ 현재 팀에서 사용하는 상태 관리 라이브러리가 있나요?

- Recoil 선택기준

1. 보일러플레이트 코드가 너무 많지 않으면 좋겠다.
2. 코드를 하나 작성하는 데 너무 많은 게 들어가지 않으면 좋겠다.
  - **단점** 비동기 업데이트 로직과 무관하게 실행하고 싶을 때 지원이 부족하여 활용이 어렵다.

즉, React의 함수 컴포넌트 혹은 Recoil의 상태 혹은 동일하게 동작하여 비동기적으로 상태 업데이트가 일어나기 때문에, 이를 즉시(동기적으로) 처리하기 어렵다.
- Redux는 하나의 상태를 관리하기 위해 너무 많은 코드 작성과, 비동기 처리를 추가하면 상태 관리가 복잡해진다.

### ▼ 상태 관리 라이브러리가 왜 필요할까요?

- 프론트 앱을 위해 컴포넌트 구조를 설계하고 만든다. 이때 계단식 형태의 트리가 생기는데, 이 컴포넌트의 리액트적인 **동작을 위한 구조**와 **데이터의 흐름이 만들어내는 구조**가 항상 일치하지 X는다. 이러한 불일치를 해소하기 위한 것이 상태 관리 라이브러리이다.
- 리엑트는 애초에 UI를 그리는 데에만 관심이 있는 거고, 데이터를 컨트롤하는 것에는 관심이 없으니까 필요하다.
- 가장 대표적으로는 Prop Drilling을 피하기 위해 사용한다.

### ▼ 상태 관리 라이브러리를 바꾼 경험이 있나요?

**메이팅** 현재 한 프로젝트에서 Redux를 쓰고 있는데 Recoil로 전환을 고려하고 있어요. 메타 정보를 담고 있는 상태의 사이즈가 커짐에 따라 작은 변경에도 상태 전체를 복사해서 업데이트하기에는 소요되는 비용이 크다고 생각해요. 하지만 Recoil의 Atom Family를 활용하여 하나의 거대한 상태가 아닌 메타 정보 내 있는 각각의 블록 정보를 개별 상태로 두고 효율적으로 상태 관리를 하고 싶어요.

**메이팅** 저는 Redux가 더 익숙한 것 같아요. reducer라는 순수 함수 기반으로 값을 조작할 수 있어서 걱정할 필요 없다는 점이 매력적입니다. 그리고 많은 사람이 사용해왔기 때문에 레퍼런스가 풍부하다는 점도 좋아요. 보일러플레이트가 약간 있는 편이지만 한 번 작성하고 나면 코드 구조가 잘 정리되어 자리 잡게 되니 괜찮은 것 같아요.