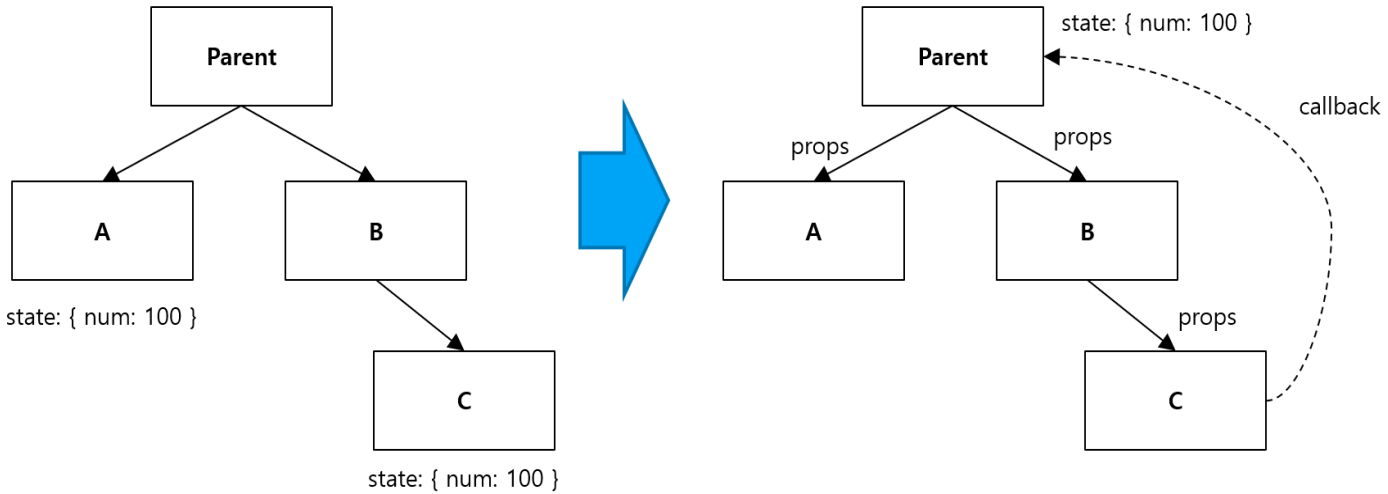
상태 관리 – Redux

1. 리액트의 상태 관리 리뷰

❖상태, 상태 변경 기능을 부모 컴포넌트에 집중

- 자식 컴포넌트에서 이벤트 발생 --〉 속성으로 전달받은 메서드 호출 --〉 부모의 상태 변경



- 부모 컴포넌트의 상태 변경 과정만 추적하면 UI를 예측할 수 있음
- 하지만 대규모 애플리케이션에서는 이 방법을 사용하기 힘듬
  - 수백개의 화면 --〉 복잡한 상태 데이터
  - 상태를 속성-속성-속성-속성-.... --〉 props drilling 해야 함
- 이러한 이유로 애플리케이션 수준의 상태 관리 기능이 필요함

2. 리액트에서 사용할 수 있는 상태관리 라이브러리

❖다양한 선택지
- Redux
- Mobx
- Recoil
- Zustand

❖이들 중에서 내가 마음에 드는 것을 사용하면 되나?
- 그럴리가... 기술, 라이브러리 스택의 선택은 팀리더 또는 PM이 결정함
- 어느 것이든 사용할 수 있는 준비가 되어 있어야 함
  - 아키텍처의 이해가 필수임.
  - 기계적으로 작성하는 것은 의미 없음

❖현재 가장 많이 쓰이는 것은 Redux
- 아키텍처의 이해가 어렵지만 장점도 많음
- Redux만 이해할 수 있다면 나머지 라이브러리는 쉽게 사용할 수 있음

3. Redux 소개

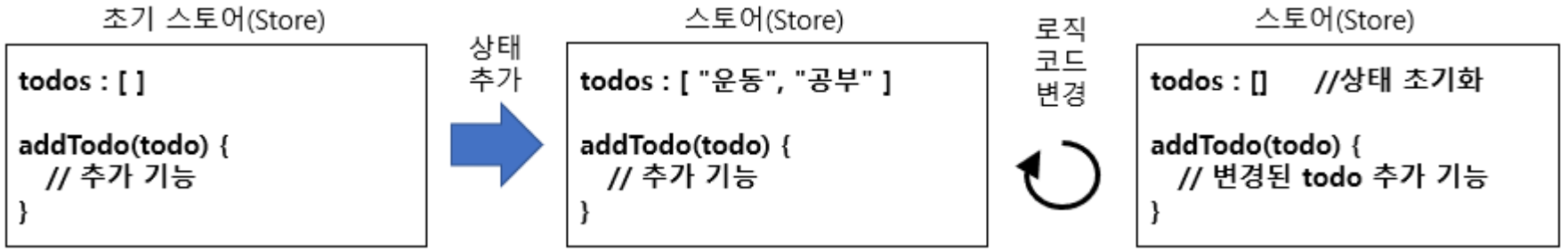
❖Redux?

- Dan Abramov
- JS 앱을 위한 예측가능한 상태 관리 컨테이너
- JS 앱에서 UI상태, 데이터 상태를 관리하기 하기 위한 도구
- Flux의 아키텍처를 발전시키면서 복잡성을 줄임
- React에서만 사용하는 것이 아님.
  - jQuery, Angular, Vue.js 에서도 사용할 수 있음.
- Redux가 제공하는 기능
  - Flux 기능 +
  - Hot Reloading +
  - 시간 여행 디버깅(Time Travel Debugging)

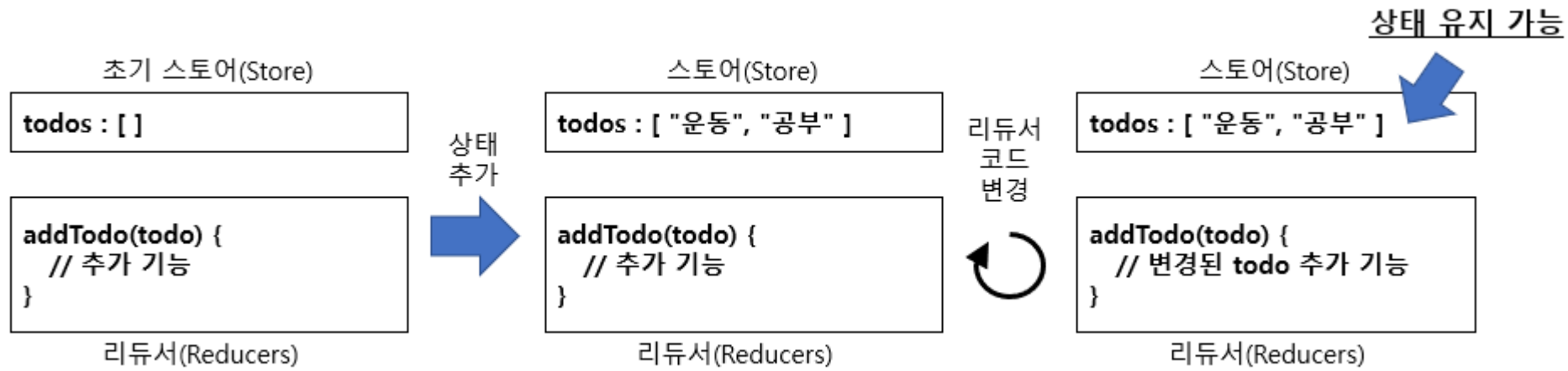
3. Redux 소개

❖Redux 특징 1

- 다른 상태 관리 라이브러리의 Store : 상태 + 상태 변경 로직
  - Store의 코드는 상태를 삭제하지 않고는 Hot Reloading이 불가능하다.



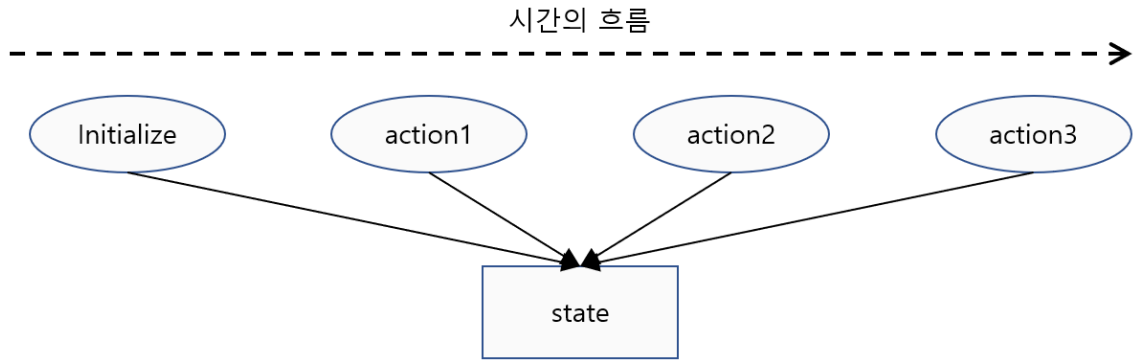
- Redux : 상태와 상태 변경 기능(Reducer)을 분리



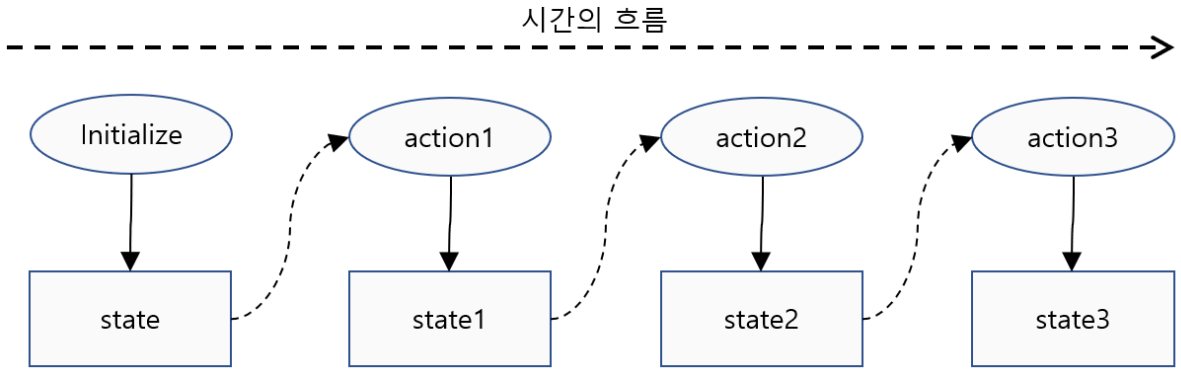
3. Redux 소개

❖Redux 특징 2

- 다른 상태 관리 라이브러리의 상태 변경 : 불변성이 필수가 아닌 경우가 많음

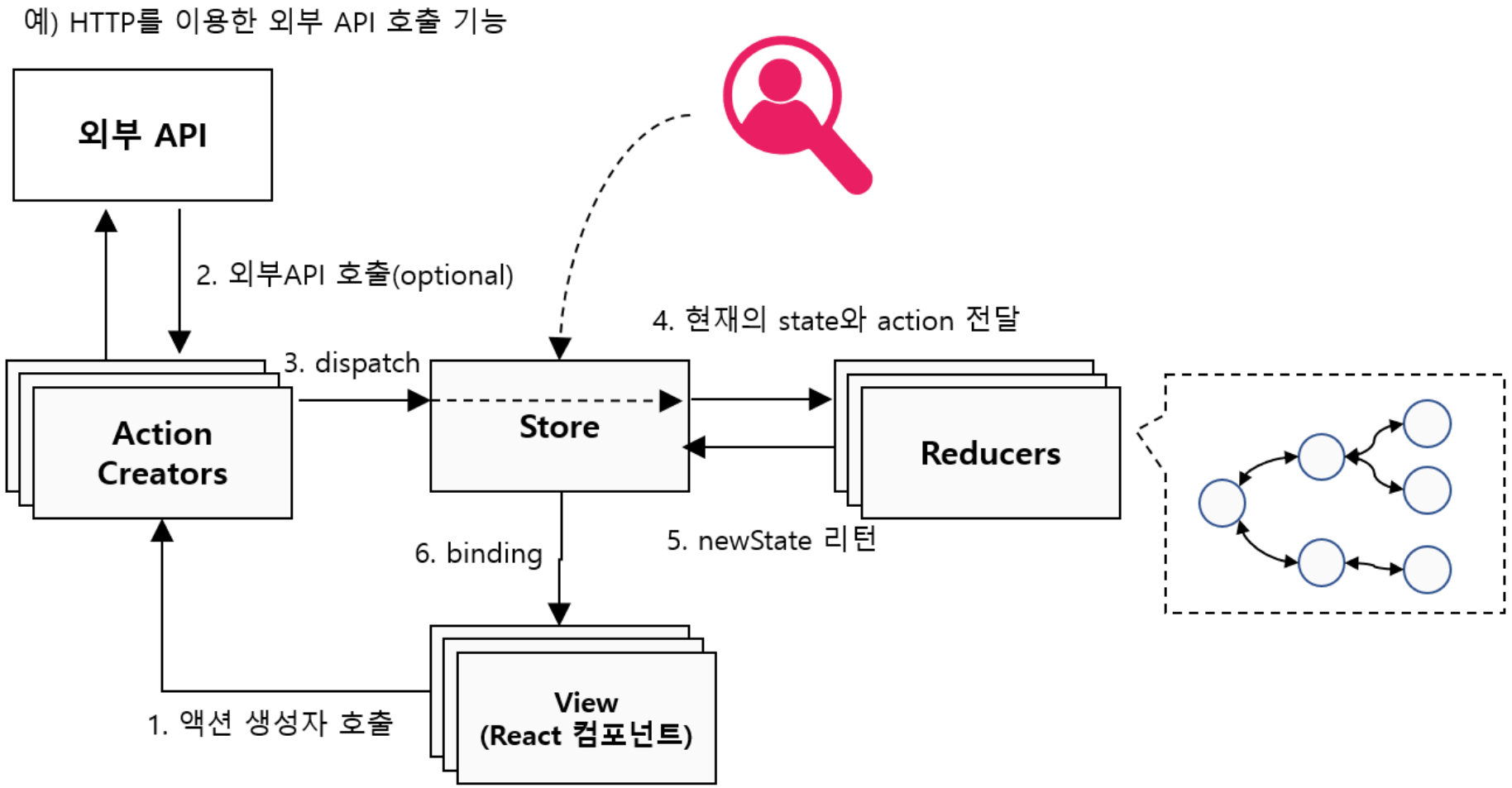


- Redux의 상태 변경 : 불변성 필수 --〉 상태 변경 추적 --〉 시간여행 디버깅!!



3. Redux 소개

❖ Redux 아키텍처



3. Redux 소개

❖리덕스 구성 요소

- 스토어(Store)
  - 단일 스토어 : 내부 상태는 읽기 전용(read only)
  - 모든 액션은 이 지점을 거쳐감
  - 이 지점만 관찰하면 상태 변경 이력, 데이터 흐름 등 상태 추적에 필요한 모든 중요한 정보를 획득할 수 있음
  - 애플리케이션 전체의 상태를 한 곳에서 관리하므로....
    - 상태(State)가 복잡해지고...
    - 상태를 변경하는 작업도 복잡해지고...
    - 따라서 상태만 스토어에서 관리! 상태 변경 작업은 리듀서에게 위임!
- 리듀서(Reducer)
  - 다중 리듀서 --〉 계층적으로 구성해야 함, 상태 트리 설계가 아주 중요함
  - 리듀서는 순수 함수
    - 입력인자가 동일하면 리턴값도 동일해야 함
    - 부작용(side effect)이 없어야 함. 외부의 값을 이용하거나 외부에 영향을 줄 수 없음
    - 함수에 전달된 인자는 불변성으로 여겨짐. 인자는 변경할 수 없음
  - 가장 대표적인 순수함수 : Array의 reduce 메서드!

3. Redux 소개

- 리듀서(이어서)

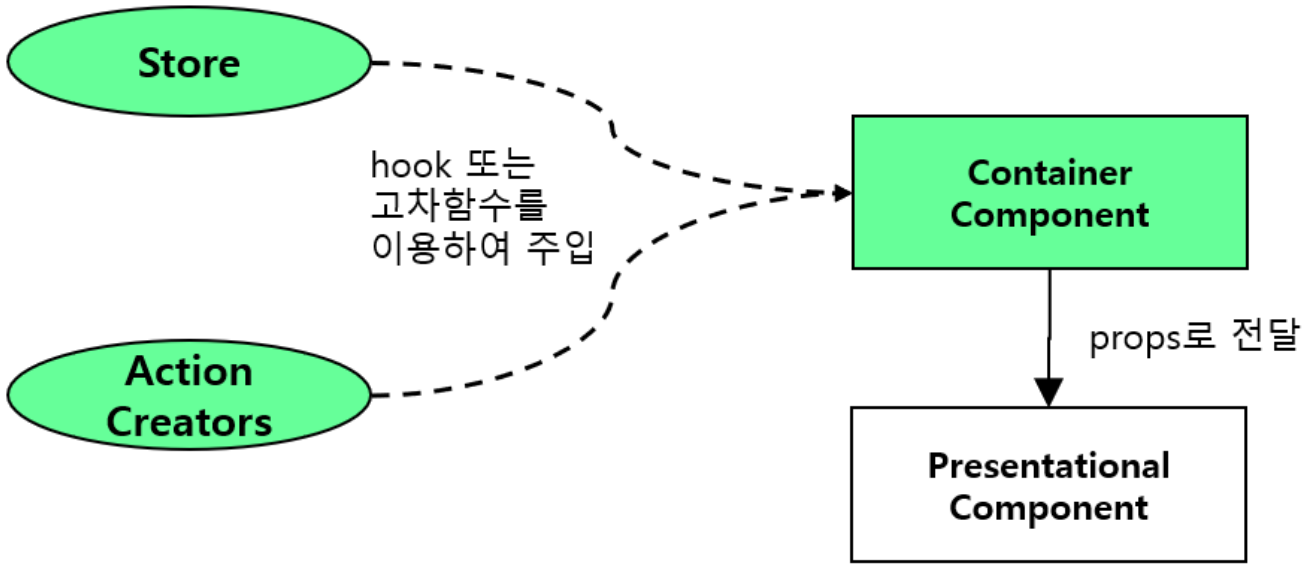
```
//자바스크립트 배열의 reduce 메서드가 사용하는 reducer 함수의 형태
//배열값들의 누적값을 구하는 예시
(accumulator, value) => {
    //새로운 값을 생성해 리턴 ---> 새로운 accumulator
   return accumulator + value;
}

//Redux의 reducer 함수
(state, action) => {
   //기존 상태와 action의 정보를 이용해 새로운 상태(newState)를 리턴 --> 새로운 상태
   return newState;
}
```

- 액션 생성자(Action Creators)
  - 액션(Action)을 생성하는 역할
  - 액션 : 상태를 변경하기 위해 전달하는 객체형태의 메시지
    - { type: "addTodo", payload : { id:1, todo:"야구 경기 관전" } }

3. Redux 소개

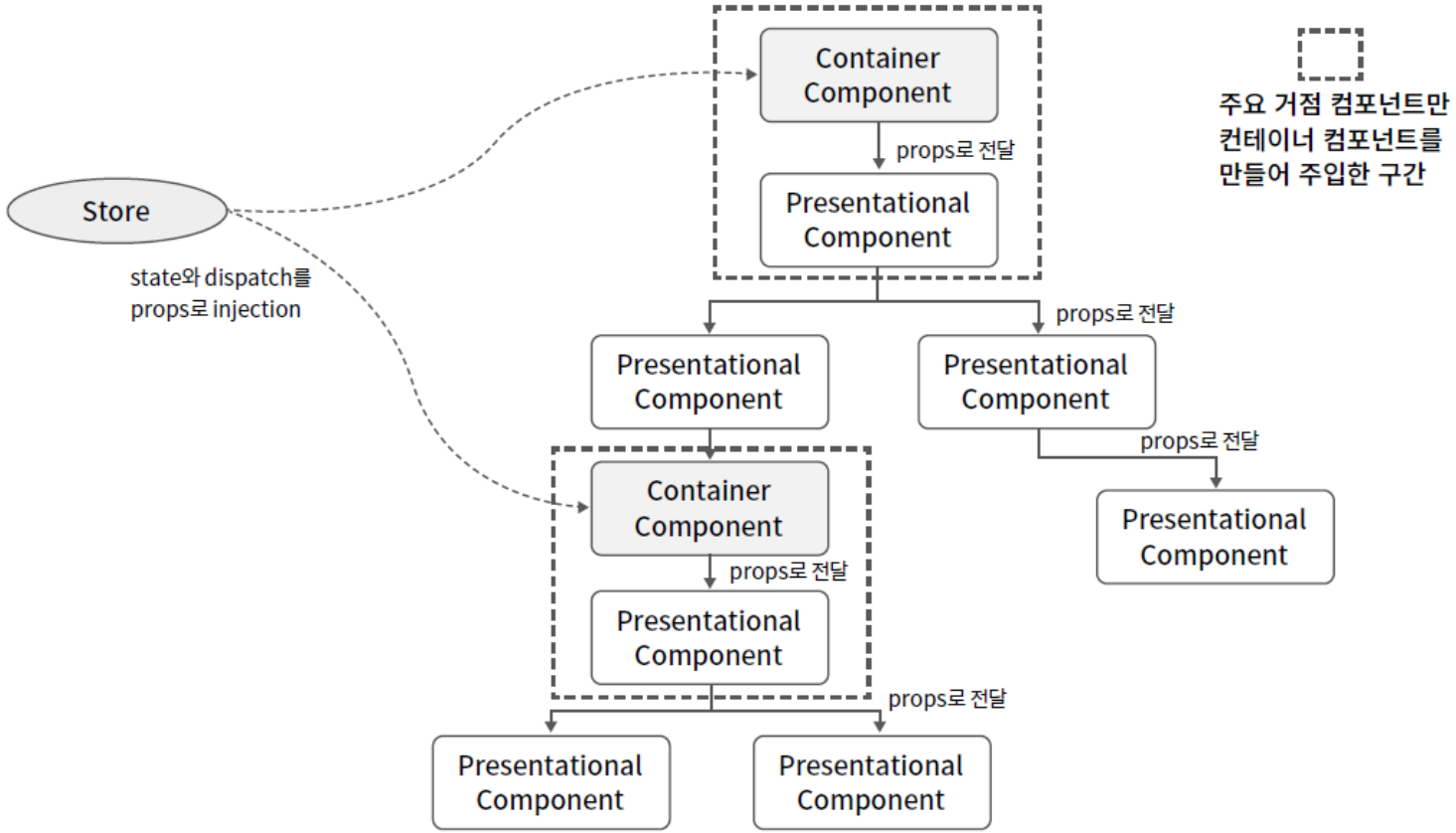
❖ Redux 컨테이너 컴포넌트

- 스토어와 연결되는 컴포넌트는 표현 컴포넌트(Presentation Component)
- 표현 컴포넌트에 스토어의 상태와 액션을 전달해주는 기능을 주입(Inject)할 수 있는 컨테이너 컴포넌트를 생성해야 함
  - react-redux 라이브러리가 제공하는 고차함수 : connect() 고차함수
  - react-redux 라이브러리가 제공하는 훅 : useSelector() 등



3. Redux 소개

- 모든 표현 컴포넌트에 대해 컨테이너 컴포넌트를 생성할까?
  - No! 주요 거점 컴포넌트에 대해서만 컨테이너 컴포넌트 작성 --〉 짧은 구간은 속성으로 전달하도록...
  - 주요 거점 컴포넌트 : 소규모 메뉴, 화면 또는 화면 레이아웃의 최상위 컴포넌트
  - 재사용성 고려



3. Redux 소개

❖ react-redux가 제공하는 훅

- 훅을 사용하는 것이 더 직관적으로 느껴짐
- useStore()
  - Redux Store 객체를 리턴함.
  - Store의 상태를 직접 읽어내려면 Store 객체의 getState() 게터 함수를 이용함
- useDispatch()
  - Redux Store 객체의 dispatch() 함수를 리턴함
  - dispatch() 함수를 이용해 액션 메시지를 스토어로 전달할 수 있음
- useSelector()
  - Store의 특정 상태를 리턴함
  - Store 상태 중에서 컴포넌트의 속성으로 전달할 필요한 상태만 받아올 수 있음

4. 미리 제공되는 예제

❖todolist-app-router-1-시작

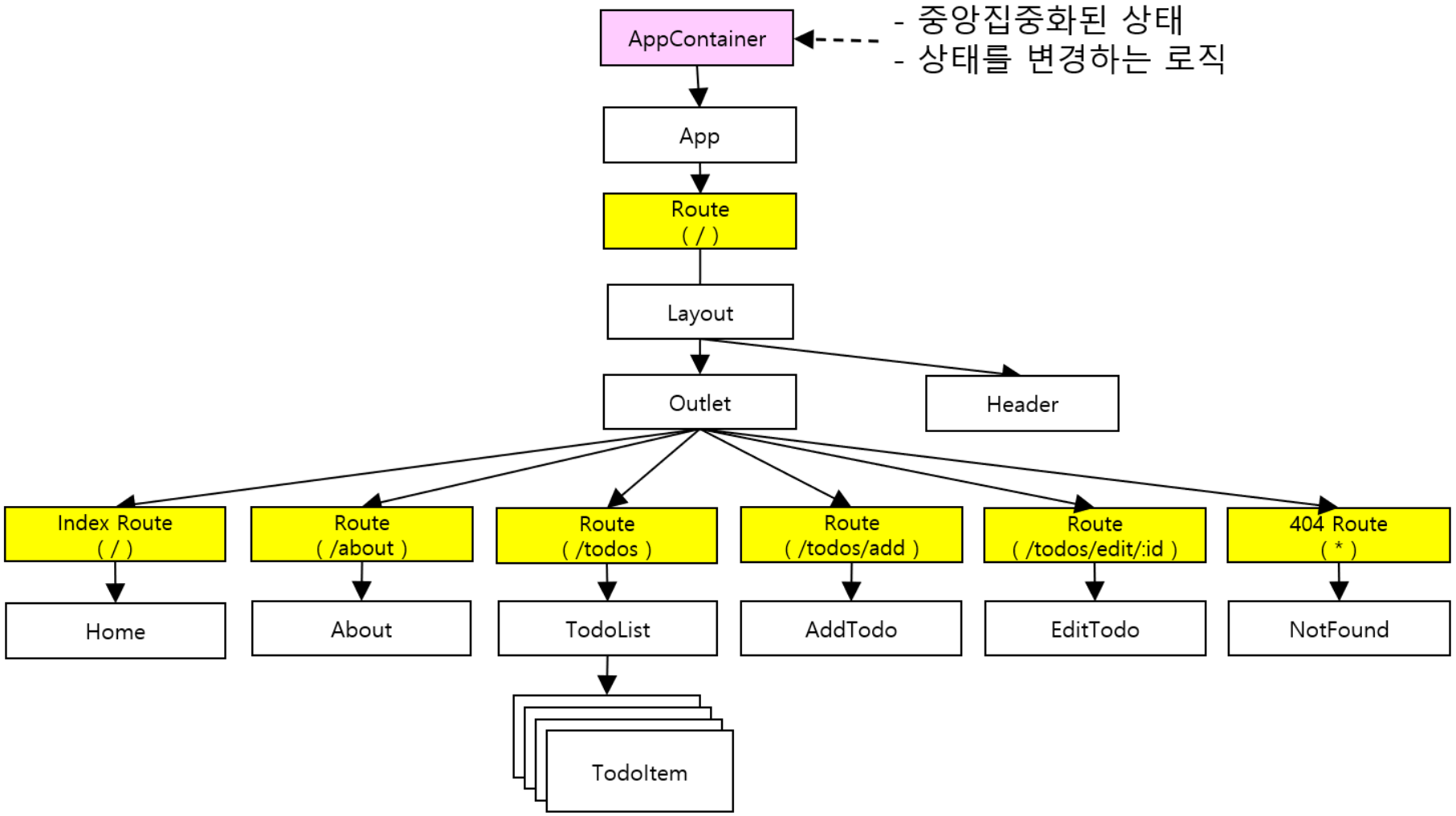
- react-router가 적용된 예제
- 상태 관리는 최상위 컴포넌트인 AppContainer가 모두 처리함
- props drilling 기법
  - 필요로 하는 자식 컴포넌트로 속성을 전달하는 방법

❖처음부터 작성하지 않고 기존 예제를 변경해보는 이유

- Redux를 사용하지 않았을 때와 사용했을 때의 차이를 비교 --〉 학습 효과 증대
- 1단계에는 Redux Toolkit 사용 배제
  - Redux Toolkit 편하지만 아키텍처를 이해하는 데에는 문제점이 있음 --〉 너무 높은 추상화
- 2단계에서 Redux Toolkit 적용

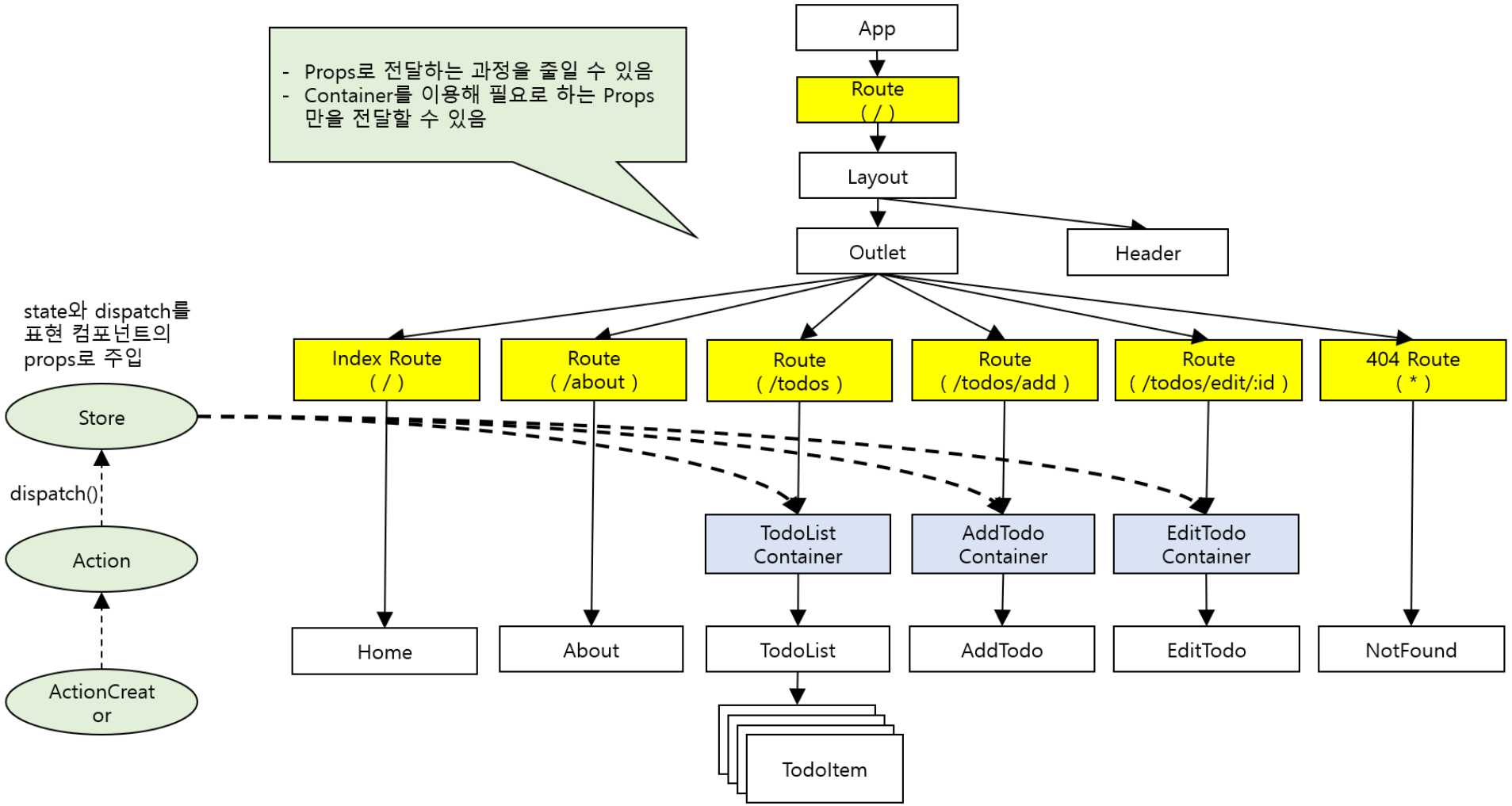
4. 미리 제공되는 예제

❖제공 예제 아키텍처



4. 미리 제공되는 예제

❖변경하려는 Redux 적용 아키텍처



5. 1단계 - Redux 적용

❖프로젝트 설정과 설계

- 패키지 설치
  - npm install redux react-redux @reduxjs/toolkit

❖상태 트리와 상태 변경 기능 설계

- 상태
  - 이 애플리케이션에서 전역 수준에서 관리할 상태는 todoList 데이터임.
  - 기존 컴포넌트에서 사용하던 모든 상태를 전역수준으로 관리할 필요 없음
- 전역 수준으로 관리할 필요가 없는 상태
  - 특정 컴포넌트에서만 사용되는 상태
  - 상태 변경을 추적할 필요가 없는 중요하지 않은 상태
  - 컴포넌트의 생명주기가 바뀌더라도(예를 들면 언마운트-마운트) 데이터가 유지될 필요가 없는 상태
- 사용할 액션(상태 변경 기능)
  - 상태가 바뀌는 작업으로 한정함
  - 이 애플리케이션에서는 4가지 상태 변경 작업(다음 페이지)

5. 1단계 - Redux 적용

- 이 애플리케이션의 상태 변경 기능 4가지

액션명	액션 설명과 액션 객체 형식
addTodo	새로운 할일을 추가 ex) { type: "addTodo", payload: { todo: string, desc: string } }
deleteTodo	기존 할일을 id로 찾아서 삭제 ex) { type: "deleteTodo", payload: { id: number } }
toggleDone	기존 할일을 id로 찾아서 완료여부(done) 필드의 true/false 값을 토글함 ex) { type: "toggleDone", payload: { id: number } }
updateTodo	기존 할일을 id로 찾아서 할일의 내용을 변경함 { type: "updateTodo", payload: { id: number, todo: string, desc: string, done: boolean } }

5. 1단계 - Redux 적용

❖ 액션 생성자 작성 : src/redux/TodoActionCreator.ts

- 액션 생성자 : Action 메시지 객체를 생성하여 리턴하는 함수

```
export const TODO_ACTION = {
  ADD_TODO: "addTodo" as const,
  DELETE_TODO: "deleteTodo" as const,
  TOGGLE_DONE: "toggleDone" as const,
  UPDATE_TODO: "updateTodo" as const,
};

const TodoActionCreator = {
  addTodo: (todoItem: { todo: string; desc: string }) => {
    return { type: TODO_ACTION.ADD_TODO, payload: todoItem };
  },
  deleteTodo: (todoItem: { id: number }) => {
    return { type: TODO_ACTION.DELETE_TODO, payload: todoItem };
  },
  toggleDone: (todoItem: { id: number }) => {
    return { type: TODO_ACTION.TOGGLE_DONE, payload: todoItem };
  },
  updateTodo: (todoItem: { id: number; todo: string; desc: string; done: boolean }) => {
    return { type: TODO_ACTION.UPDATE_TODO, payload: todoItem };
  },
};
```

5. 1단계 - Redux 적용

❖액션 생성자 작성 : src/redux/TodoActionCreator.ts (이어서)

```
//각 액션생성자 함수의 리턴값 타입을  Union 하여 액션 타입 선언
export type TodoActionType =
  | ReturnType<typeof TodoActionCreator.addTodo>
  | ReturnType<typeof TodoActionCreator.deleteTodo>
  | ReturnType<typeof TodoActionCreator.toggleDone>
  | ReturnType<typeof TodoActionCreator.updateTodo>;

export default TodoActionCreator;
```

5. 1단계 - Redux 적용

❖리듀서 작성 : src/redux/TodoReducer.ts

```
import { produce } from "immer";
import { TodoActionType, TODO_ACTION } from "./TodoActionCreator";

export type TodoItemType = {
  id: number;
  todo: string;
  desc: string;
  done: boolean;
};

export type TodoStatesType = { todoList: TodoItemType[] };

const initialState: TodoStatesType = {
  todoList: [
    { id: 1, todo: "ES6학습", desc: "설명1", done: false },
    { id: 2, todo: "React학습", desc: "설명2", done: false },
    { id: 3, todo: "ContextAPI 학습", desc: "설명3", done: true },
    { id: 4, todo: "야구경기 관람", desc: "설명4", done: false },
  ],
};
```

5. 1단계 - Redux 적용

❖리듀서 작성 : src/redux/TodoReducer.ts(이어서)

```
const TodoReducer = (state: TodoStatesType = initialState, action: TodoActionType) => {
  let index: number;
  switch (action.type) {
    case TODO_ACTION.ADD_TODO:
      return produce(state, (draft) => {
        draft.todoList.push({ id: new Date().getTime(), todo: action.payload.todo, desc: action.payload.desc, done: false });
      });
    case TODO_ACTION.DELETE_TODO:
      return state.todoList.filter((item) => item.id !== action.payload.id);
    case TODO_ACTION.TOGGLE_DONE:
      index = state.todoList.findIndex((item) => item.id === action.payload.id);
      return produce(state, (draft) => {
        draft.todoList[index].done = !draft.todoList[index].done;
      });
    case TODO_ACTION.UPDATE_TODO:
      index = state.todoList.findIndex((item) => item.id === action.payload.id);
      return produce(state, (draft) => {
        draft.todoList[index] = { ...action.payload };
      });
    default:
      return state;
  }
};

export default TodoReducer;
```

5. 1단계 - Redux 적용

❖리듀서

- 애플리케이션을 처음 실행했을 때는 Store가 비어있음
- 이것을 채워주기 위해 초기 상태(initialState)가 필요
  - 처음으로 리듀서가 호출될 때 state는 undefined 전달
  - 이 때 initialState가 주어지고 switch 문의 default case를 통해 초기 상태가 스토어로 전달됨

```
const TodoReducer = (state: TodoStatesType = initialState, action: TodoActionType) => {
  ......
}
```

  - 따라서 switch 문에는 반드시 default case가 기존 상태를 리턴하도록 작성해야 함.
- 모든 변경은 불변성을 가지도록 해야 함

5. 1단계 - Redux 적용

❖ 스토어 작성 : src/redux/AppStore.ts

- redux 가 제공하는 createStore 함수를 이용할 수 있지만 최근에는 reduxjs toolkit이 제공하는 configureStore을 더 많이 사용
  - createStore() : deprecated
- @reduxjs/toolkit
  - 리듀서, 스토어를 작성할 때 사용할 수 있는 여러가지 툴킷 함수를 제공
  - 더 간단하고 정리된 코드로 작성할 수 있음
    - 그러나 처음에는 툴킷에 의존하지 말고 조금 어렵더라도 기본 라이브러리를 이용하는 것이 개념 이해에 도움이 됨

```
import { configureStore } from "@reduxjs/toolkit";
import TodoReducer from "./TodoReducer";

const AppStore = configureStore({ reducer: TodoReducer });
export default AppStore;
```

5. 1단계 - Redux 적용

❖src/main.jsx 변경

- Provider를 통해 store를 제공해야 함
- AppContainer는 더이상 사용하지 않음

```
import React from "react";
import ReactDOM from "react-dom/client";
import "bootstrap/dist/css/bootstrap.css";
//import AppContainer from "./AppContainer";
import "./index.css";
import App from "./App";

import AppStore from "./redux/AppStore";
import { Provider } from "react-redux";

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

5. 1단계 - Redux 적용

❖각 컴포넌트별로 컨테이너 컴포넌트 작성

- react-redux가 제공하는 useSelector(), useDispatch() 훅 사용

- App 컴포넌트
  - 더이상 속성이 필요하지 않음
    - 속성 제거
    - 자식 컴포넌트로 다시 속성을 전달하는 부분도 제거
- TodoList 컴포넌트
  - TodoListContainer 작성
  - useDispatch() 훅으로 받아낸 dispatch 함수를 이용해 Action 메시지 객체 전송
    - dispatch(TodoActionCreator.deleteTodo({ id }))
      --〉 dispatch({ type: "deleteTodo", payload : { id: id } })
  - useSelector()훅을 이용해 스토어의 상태 중 todoList필요한 것만 속성으로 전달하는

5. 1단계 - Redux 적용

- AddTodo 컴포넌트
  - 속성 : addTodo
  - useDispatch() 훅을 이용해 액션을 전달하도록 AddTodoContainer 작성
- EditTodo 컴포넌트 변경
  - 속성 : updateTodo, todoList
  - EditTodoContainer 작성
    - useSelector()훅을 이용해 스토어의 상태 중 todoList를 속성으로 전달
    - useDispatch() 훅을 이용해 updateTodo 액션 전달 메서드를 속성으로 전달

5. 1단계 - Redux 적용

❖ 예제 : App.tsx 변경

```
import { BrowserRouter as Router, Route, Routes } from "react-router-dom";
import Layout from "./components/Layout";

import Home from "./pages/Home";
import About from "./pages/About";
import TodoList from "./pages/TodoList";
import AddTodo from "./pages/AddTodo";
import EditTodo from "./pages/EditTodo";
import NotFound from "./pages/NotFound";

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="about" element={<About />} />
          <Route path="todos" element={<TodoList />} />
          <Route path="todos/add" element={<AddTodo />} />
          <Route path="todos/edit/:id" element={<EditTodo />} />
          <Route path="*" element={<NotFound />} />
        </Route>
      </Routes>
    </Router>
  );
};

export default App;
```

5. 1단계 - Redux 적용

❖ 예제 : TodoList.tsx 변경

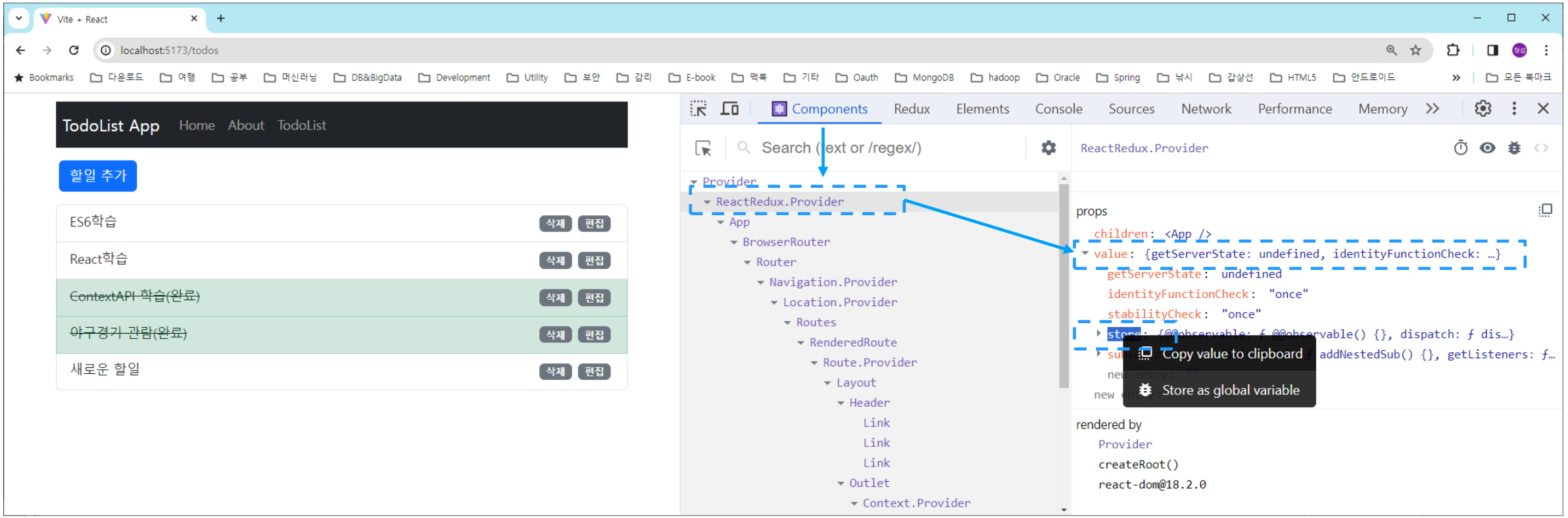
```
import { Link } from "react-router-dom";
import TodoItem from "./TodoItem";
import { useDispatch, useSelector } from "react-redux";
import { TodoItemType } from "../redux/TodoReducer";
import TodoActionCreator from "../redux/TodoActionCreator";

type PropsType = {
  todoList: TodoItemType[];
  deleteTodo: ({ id }: { id: number }) => void;
  toggleDone: ({ id }: { id: number }) => void;
};

const TodoList = ({ todoList, deleteTodo, toggleDone }: PropsType) => {
  .....(생략)
};

const TodoListContainer = () => {
  const dispatch = useDispatch();
  const todoList = useSelector((state: { todoList: TodoItemType[] }) => state.todoList);
  const toggleDone = (args: { id: number }) => dispatch(TodoActionCreator.toggleDone(args));
  const deleteTodo = (args: { id: number }) => dispatch(TodoActionCreator.deleteTodo(args));
  return <TodoList todoList={todoList} deleteTodo={deleteTodo} toggleDone={toggleDone} />;
};

export default TodoListContainer;
export { TodoList };
```

5. 1단계 - Redux 적용

❖AddTodo, EditTodo, TodoItem 컴포넌트

- 완성예제로부터 Copy & Paste 후 검토

❖실행 결과



6. 2단계 - Redux Toolkit 적용

❖1단계 예제의 문제점

- 순수하게 Redux만 사용한 예제
- 아키텍처 이해에는 도움이 되지만 몇가지 어려움이 존재함
  - 리듀서의 문제점
    - 순수함수, 불변성을 반드시 사용 : immer와 같은 라이브러리를 이용하여 새로운 상태를 만들어 리턴해야 함
    - default case를 반드시 지정해야 함 : Action의 type이 일치하는 것이 없는 경우, 기존 상태를 리턴해야 함
  - 액션 생성자의 문제점
    - 오류 방지 목적으로 상수를 정의해야 함
    - 액션 메시지를 직접 만들어 리턴해야 함

❖리덕스 툴킷(@reduxjs/toolkit)

- 다양한 헬퍼 라이브러리, 함수들 제공
- createAction, createReducer, createSlice, configureStore 등

6. 2단계 - Redux Toolkit 적용

❖createAction()

- 액션 생성자를 간단하게 생성할 수 있도록 도와줌
- 상수 정의를 하지 않아도 됨 : 액션생성자 자체가 상수를 제공함
- createAction<T>(actionName:string)

```
//기존 예제 : RTK 사용(X)
export const TodoActionCreator = {
  addTodo: ({ todo, desc }: { todo:string; desc:string }) => {
    return { type: TODO_ACTION.ADD_TODO, payload: { todo, desc } };
  },
  ......
};

//변경된 예제 : RTK 사용(O)
export const TodoActionCreator = {
  addTodo: createAction<{ todo:string; desc:string }>("addTodo"),
  ......
};
```

6. 2단계 - Redux Toolkit 적용

❖createReducer()

- immer 불변성 라이브러리 기능이 내장되어 있음
  - 새로운 상태를 만들어서 리턴할 필요 없이 직접 상태 내부 데이터 변경함
- createReducer 함수의 인자
  - initialState : 초기 상태 지정
  - builder 콜백 함수 : builder 인자를 이용해 case마다의 변경기능을 추가함
  - default case를 작성할 필요 없음

```
const TodoReducer = createReducer(initialState, (builder) => {
   builder.
     .addCase(TodoActionCreator.addTodo, (state, action) => {
       //불변성을 사용하지 않고 직접 state 변경
     })
     ……
})
```

6. 2단계 - Redux Toolkit 적용

❖ TodoActionCreator.js 변경

```
import { createAction } from "@reduxjs/toolkit";

const TodoActionCreator = {
  addTodo: createAction<{ todo: string; desc: string }>("addTodo"),
  deleteTodo: createAction<{ id: number }>("deleteTodo"),
  toggleDone: createAction<{ id: number }>("toggleDone"),
  updateTodo: createAction<{ id: number; todo: string; desc: string; done: boolean }>("updateTodo"),
};

export default TodoActionCreator;
```

6. 2단계 - Redux Toolkit 적용

❖TodoReducer.ts 변경

```
import { createReducer } from "@reduxjs/toolkit";
.....(생략)


const TodoReducer = createReducer(initialState, (builder) => {
  builder
    .addCase(TodoActionCreator.addTodo, (state, action) => {
      state.todoList.push({ ...action.payload, id: new Date().getTime(), done: false });
    })
    .addCase(TodoActionCreator.deleteTodo, (state, action) => {
      const index = state.todoList.findIndex((item) => item.id === action.payload.id);
      state.todoList.splice(index, 1);
    })
    .addCase(TodoActionCreator.toggleDone, (state, action) => {
      const index = state.todoList.findIndex((item) => item.id === action.payload.id);
      state.todoList[index].done = !state.todoList[index].done;
    })
    .addCase(TodoActionCreator.updateTodo, (state, action) => {
      const index = state.todoList.findIndex((item) => item.id === action.payload.id);
      state.todoList[index] = { ...action.payload };
    });
});

export default TodoReducer;
```

6. 2단계 - Redux Toolkit 적용

❖액션 생성자, 리듀서를 한번에 만들면 안될까?

- 그래서 등장한 것이 createSlice()

❖createSlice()

- 초기 상태, 리듀서 함수, 이름으로 구성된 객체(Slice)를 옵션 인자로 전달받아 리듀서와 상태에 해당하는 액션 생성자와 액션 유형을 자동으로 생성하는 고차함수
- slice : initialState + Reducer + Name + ActionType
- "todolist-app-router-4-rtk-slice" 예제 검토할 것

❖Slice를 반드시 사용해야 하는가?

- 너무 높은 추상화는 개발자의 시야를 흐리게 함
- 라이브러리를 사용은 하되 매몰되지 않아야 함
  - 버전 바뀌면 다시 배워야 함.
- 아키텍처를 이해하는 것이 중요함

7. 다중 리듀서

❖애플리케이션의 상태가 복잡해지면?

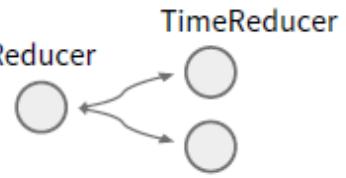
- 리듀서의 상태 변경 기능도 많아지고 복잡해짐
  - 하나의 리듀서로 처리 불가능
  - 따라서 여러 개의 리듀서(다중 리듀서)로 분리시켜야 함
- 다중 리듀서를 사용하려면...
  - 자식 리듀서들은 전체 상태 트리 중 특정 하위의 트리를 담당하기 때문에 상태 트리를 꼼꼼하게 설계해야 함

- 사용 메서드 : combineReducers()

7. 다중 리듀서

❖ 다중 리듀서와 combineReducers()

```
const RootReducer = combineReducers({
      home : TimeReducer,
      todos: TodoReducer
});
```




```
{
  "home": {
    "currentTime": "2020-04-14T06:08:10.009Z"
  },
  "todos": {
    "todolist": [
      {
        "id": 1,
        "todo": "ES6학습",
        "desc": "설명1",
        "done": false
      },
      {
        "id": 2,
        "todo": "React학습",
        "desc": "설명2",
        "done": false
      },
      {
        "id": 3,
        "todo": "ContextAPI 학습",
        "desc": "설명3",
        "done": true
      },
      {
        "id": 4,
        "todo": "야구경기 관람",
        "desc": "설명4",
        "done": false
      }
    ]
  }
}
```

TimeReducer의 state

TodoReducer의 state

7. 다중 리듀서

❖ 다중 리듀서 기능 적용

- 기능을 확인하기 위해 Todolist 예제에 새로운 컴포넌트 추가와 약간의 코드 추가
  - MyTime 컴포넌트
  - TimeReducer
  - RootReducer : TimeReducer와 TodoReducer를 결합한 Root Reducer
  - TimeActionCreator
- 추가할 상태와 Dispatch 메서드
  - currentTime, changeTime()
- EditTodo, TodoList 컴포넌트에서의 변경
  - 상태 트리가 변경되었기 때문에 주입해야 할 상태가 다름
  - const todoList = useSelector((state:RootStatesType)=>**state.todos.todoList**);

7. 다중 리듀서

❖TimeActionCreator, TimeReducer 작성

```
import { createAction } from "@reduxjs/toolkit";
const TimeActionCreator = {
  changeTime: createAction<{ currentTime: Date }>("changeTime"),
};
export default TimeActionCreator;
```

```
import { createReducer } from "@reduxjs/toolkit";
import TimeActionCreator from "./TimeActionCreator";

export type TimeStatesType = { currentTime: Date };

const initialState: TimeStatesType = {
  currentTime: new Date(),
};

const TimeReducer = createReducer(initialState, (builder) => {
  builder.addCase(TimeActionCreator.changeTime, (state, action) => {
    state.currentTime = action.payload.currentTime;
  });
});

export default TimeReducer;
```

7. 다중 리듀서

❖ AppStore 변경

```
import { configureStore } from "@reduxjs/toolkit";
import { combineReducers } from "redux";
import TimeReducer, { TimeStatesType } from "./TimeReducer";
import TodoReducer, { TodoStatesType } from "./TodoReducer";

export type RootStatesType = {
  home: TimeStatesType;
  todos: TodoStatesType;
};

const RootReducer = combineReducers({
  home: TimeReducer,
  todos: TodoReducer
});

const AppStore = configureStore({
  reducer: RootReducer,
  middleware: (getDefaultMiddleware) => {
    return getDefaultMiddleware({ serializableCheck: false });
  },
});

export default AppStore;
```

7. 다중 리듀서

❖TodoList, EditTodo 컴포넌트 변경

```
const TodoListContainer = ()=>{
  const dispatch = useDispatch();
  const todoList = useSelector((state)=>state.todos.todoList);
  const toggleDone = (id) => dispatch(TodoActionCreator.toggleDone({id}))
  const deleteTodo = (id) => dispatch(TodoActionCreator.deleteTodo({id}))
  return <TodoList todoList={todoList} deleteTodo={deleteTodo} toggleDone={toggleDone} />
}
```

```
const EditTodoContainer = () => {
  const dispatch = useDispatch();
  const todoList = useSelector((state)=>state.todos.todoList);
  const updateTodo = (id, todo, desc, done) => dispatch(TodoActionCreator.updateTodo({ id, todo, desc, done }))

  return <EditTodo todoList={todoList} updateTodo={updateTodo} />
}
```

7. 다중 리듀서

❖src/pages/MyTime.tsx 추가

```
type PropsType = {
  currentTime: Date;
  changeTime: ({ currentTime }: { currentTime: Date }) => void;
};

const MyTime = ({ currentTime, changeTime }: PropsType) => {
  return (
    <div className="row">
      <div className="col">
        <button className="btn btn-primary" onClick={() => changeTime({ currentTime: new Date() })}>
          현재 시간 확인
        </button>
        <h4>
          <span className="label label-default">{currentTime.toLocaleString()}</span>
        </h4>
      </div>
    </div>
  );
};

export default MyTime;
```

7. 다중 리듀서

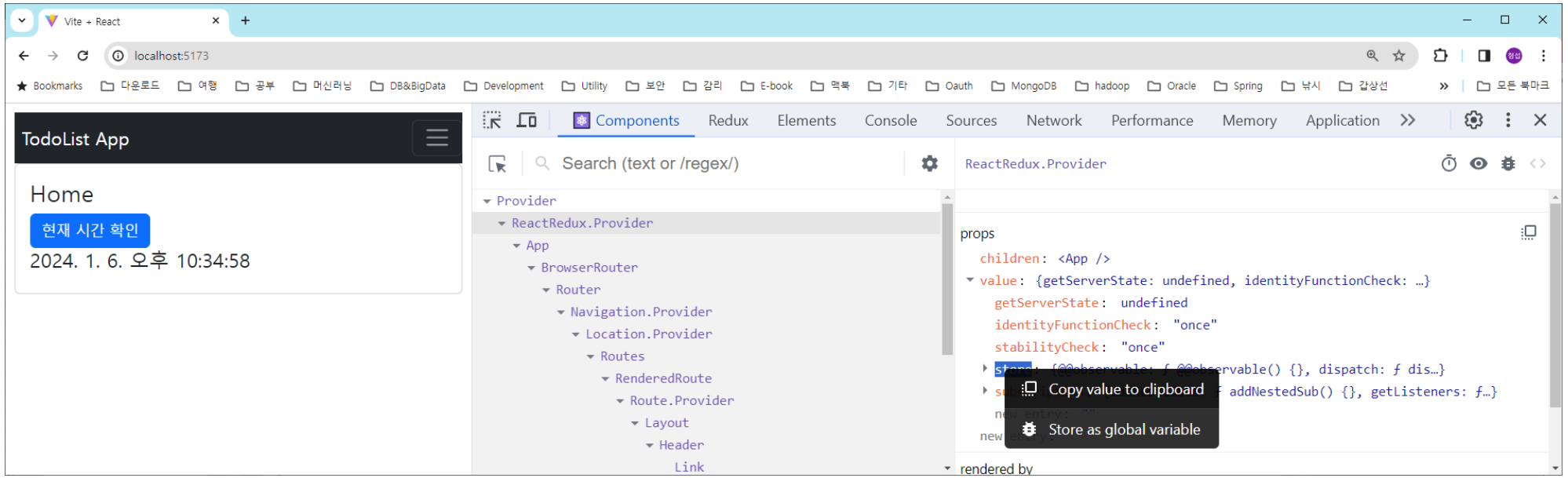
❖Home 컴포넌트 변경

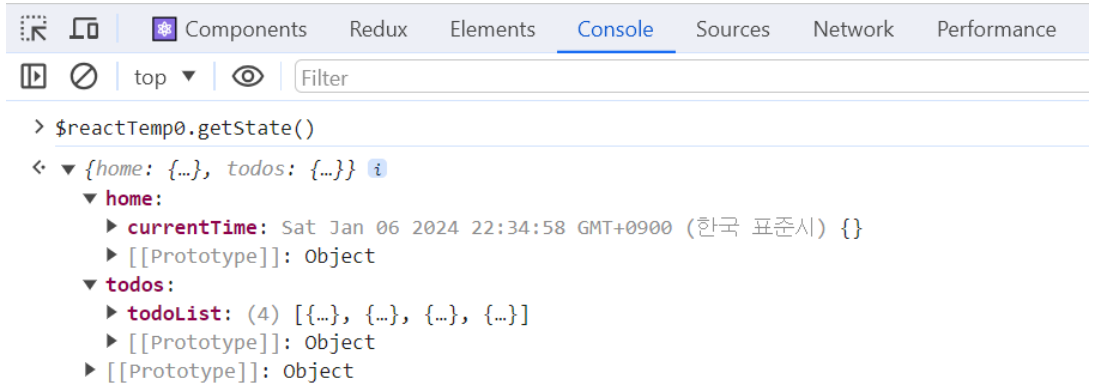
```
import { RootStatesType } from "../redux/AppStore";
import TimeActionCreator from "../redux/TimeActionCreator";
import MyTime from "./MyTime";
import { useDispatch, useSelector } from "react-redux";

type PropsType = {
  currentTime: Date;
  changeTime: ({ currentTime }: { currentTime: Date }) => void;
};
const Home = ({ currentTime, changeTime }: PropsType) => {
  return (
    <div className="card card-body">
      <h2>Home</h2>
      <MyTime currentTime={currentTime} changeTime={changeTime} />
    </div>
  );
};
const HomeContainer = () => {
  const dispatch = useDispatch();
  const currentTime = useSelector((state: RootStatesType) => state.home.currentTime);
  const changeTime = (args: { currentTime: Date }) => dispatch(TimeActionCreator.changeTime(args));
  return <Home currentTime={currentTime} changeTime={changeTime} />;
};

export default HomeContainer;
```

7. 다중 리듀서

❖실행 결과

- 전체 상태 트리 확인

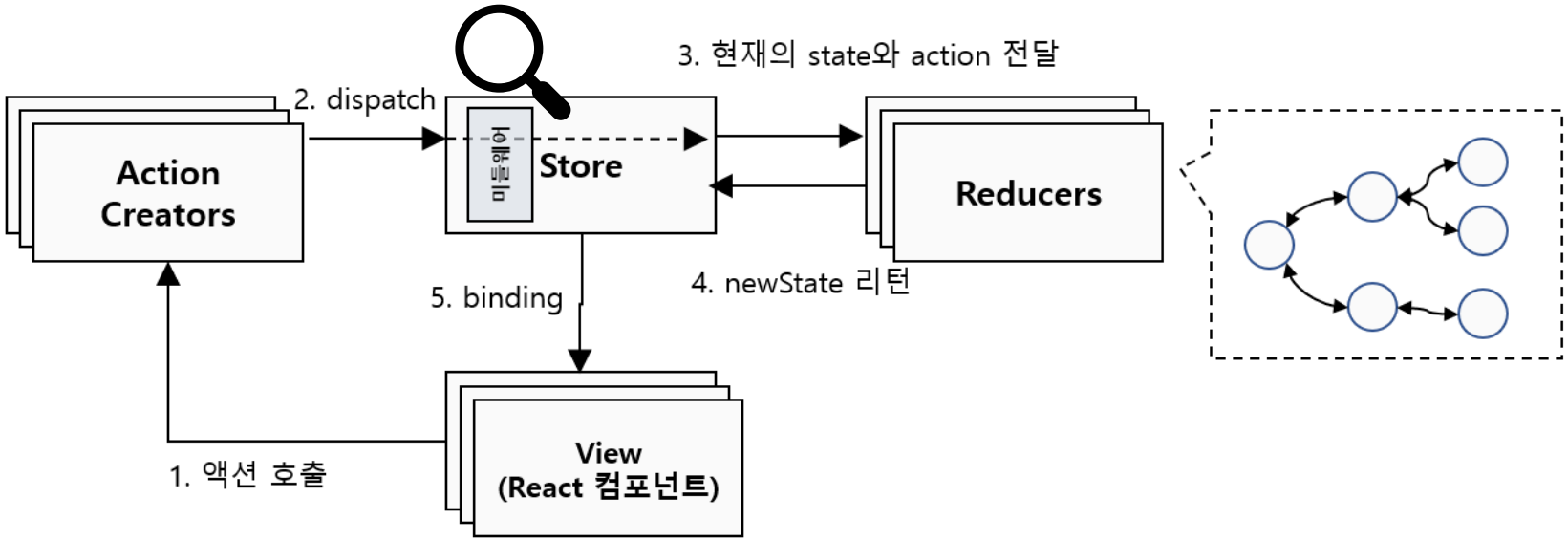




8. Redux Middleware

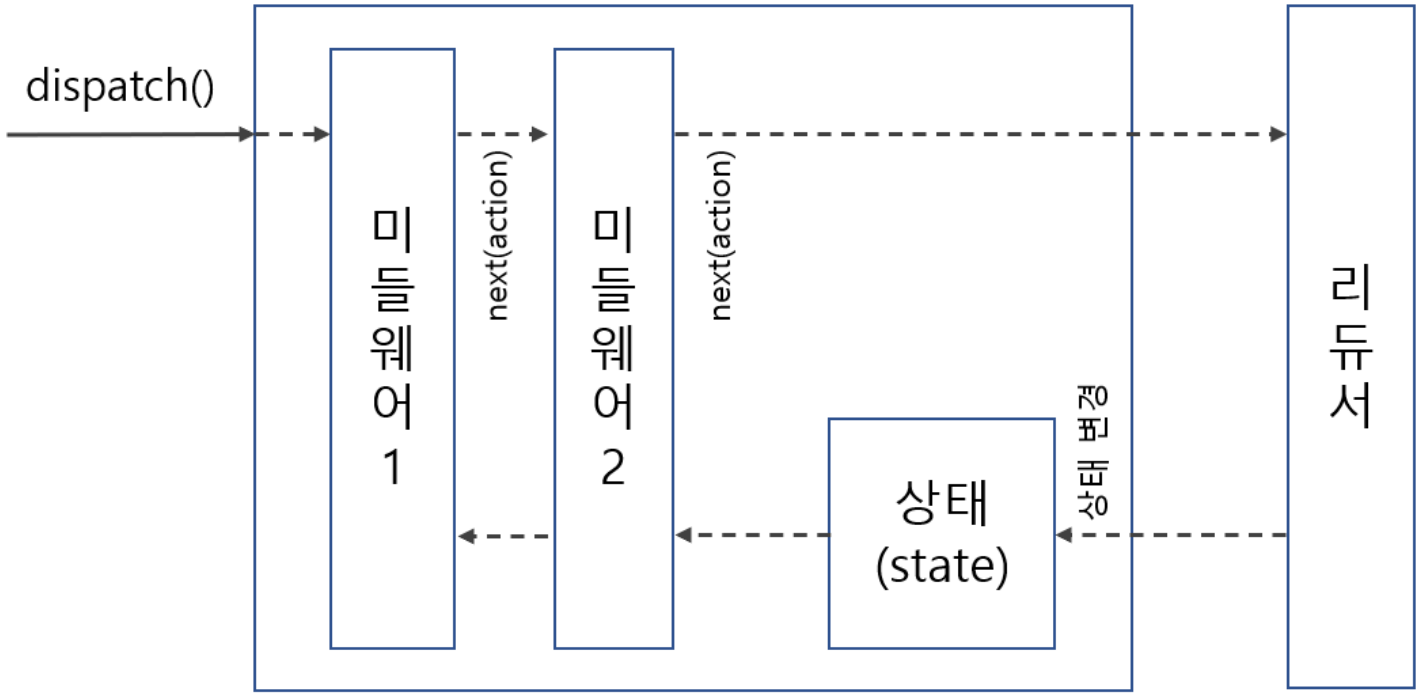
❖리덕스 미들웨어란?

- 액션이 스토어로 dispatch 된 후 리듀서에 도달하기 전과 상태 변경이 완료된 후 수행할 중앙집중화 된 작업을 지정할 수 있는 함수
- 단일 스토어 내부에 등록함
  - 모든 액션이 스토어를 거쳐감
  - 상태는 스토어에 저장



8. Redux Middleware

❖좀 더 상세하게 보자면...



- ❖ 미들웨어 1의 전처리 실행 -〉next(action)
- ❖ 미들웨어 2의 전처리 실행 -〉next(action)
- ❖ 리듀서 실행 -〉 새로운 상태 리턴
- ❖ Store의 새로운 상태 설정
- ❖ 미들웨어 2의 후처리 실행
- ❖ 미들웨어 1의 후처리 실행

8. Redux Middleware

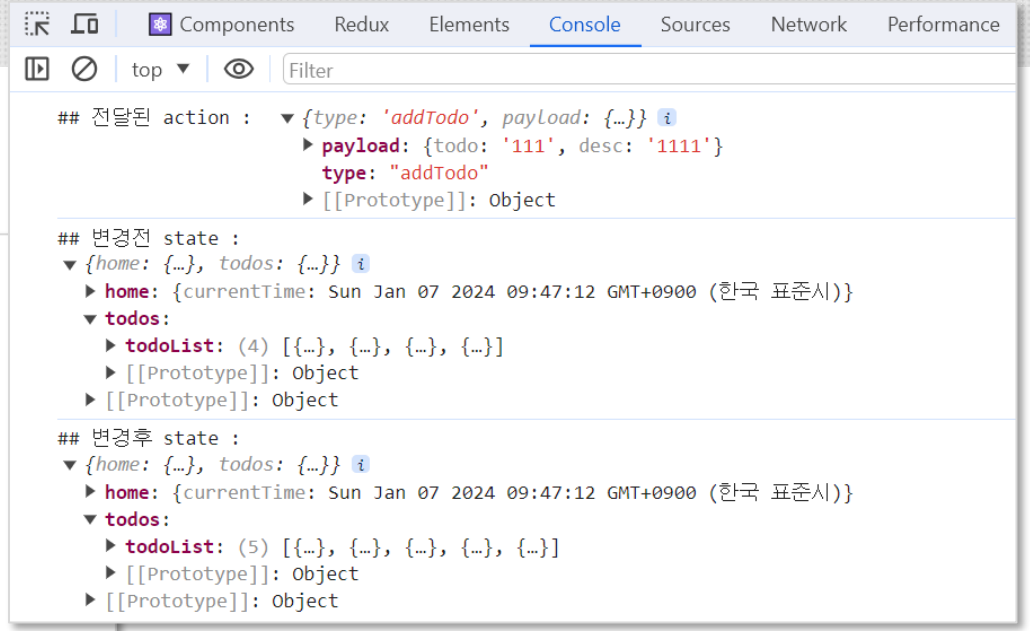
❖좀 더 상세하게 보자면...

```
// Arrow Function 사용
const middleware1: Middleware = (store)=>(next)=>(action)=> {
  //다음 미들웨어 또는 리듀서로 전달되기 전
  next(action);
  //스토어의 새로운 상태 설정 후
}

// 전통적인 Function 사용
var middleware1: Middleware = function middleware1(store) {
  return function (next) {
    return function (action) {
      //다음 미들웨어 또는 리듀서로 전달되기 전
      next(action);
      //스토어의 새로운 상태 설정 후
    };
  };
};
```

8. Redux Middleware

❖간단한 Console Logger 작성

```
……
const logger: Middleware = (store) => (next) => (action) => {
  console.log("## 전달된 action : ", action);
  console.log("## 변경전 state : ", store.getState());
  next(action);
  console.log("## 변경후 state : ", store.getState());
};

const AppStore = configureStore({
  reducer: RootReducer,
  middleware: (getDefaultMiddleware) => {
    return getDefaultMiddleware({ serializableCheck: false }).concat(logger);
  },
});

……
```



9. 미들웨어와 비동기 처리

❖복잡하고 긴 처리 시간이 필요한 작업

- 동기적으로 처리하면?
  - 처리 시간이 길어지면 그 시간 동안 브라우저가 먹통이 됨
  - 따라서 비동기적으로 처리해야 함
- 대표적인 예
  - 백엔드 API 서비스와의 통신 : 네트웍을 통해 전송되는 시간 + 백엔드에서의 실행 시간
  - setTimeout()을 이용해 일정 시간 뒤에 실행하도록 처리

❖리덕스를 사용하는 애플리케이션에서는 어느 지점에서 비동기 처리를 할까?

- 리듀서? No! 순수함수!!
- 액션 생성자
  - 좋은 위치이긴 하지만 액션 생성자는 액션(메시지 객체)을 생성하여 리턴함
  - 액션 생성자 함수는 값을 리턴하기 때문에 동기적으로 작동!
  - 이런 이유로 미들웨어를 이용해야 함
  - redux-thunk, redux-saga

10. redux-thunk 미들웨어

❖redux-thunk란?

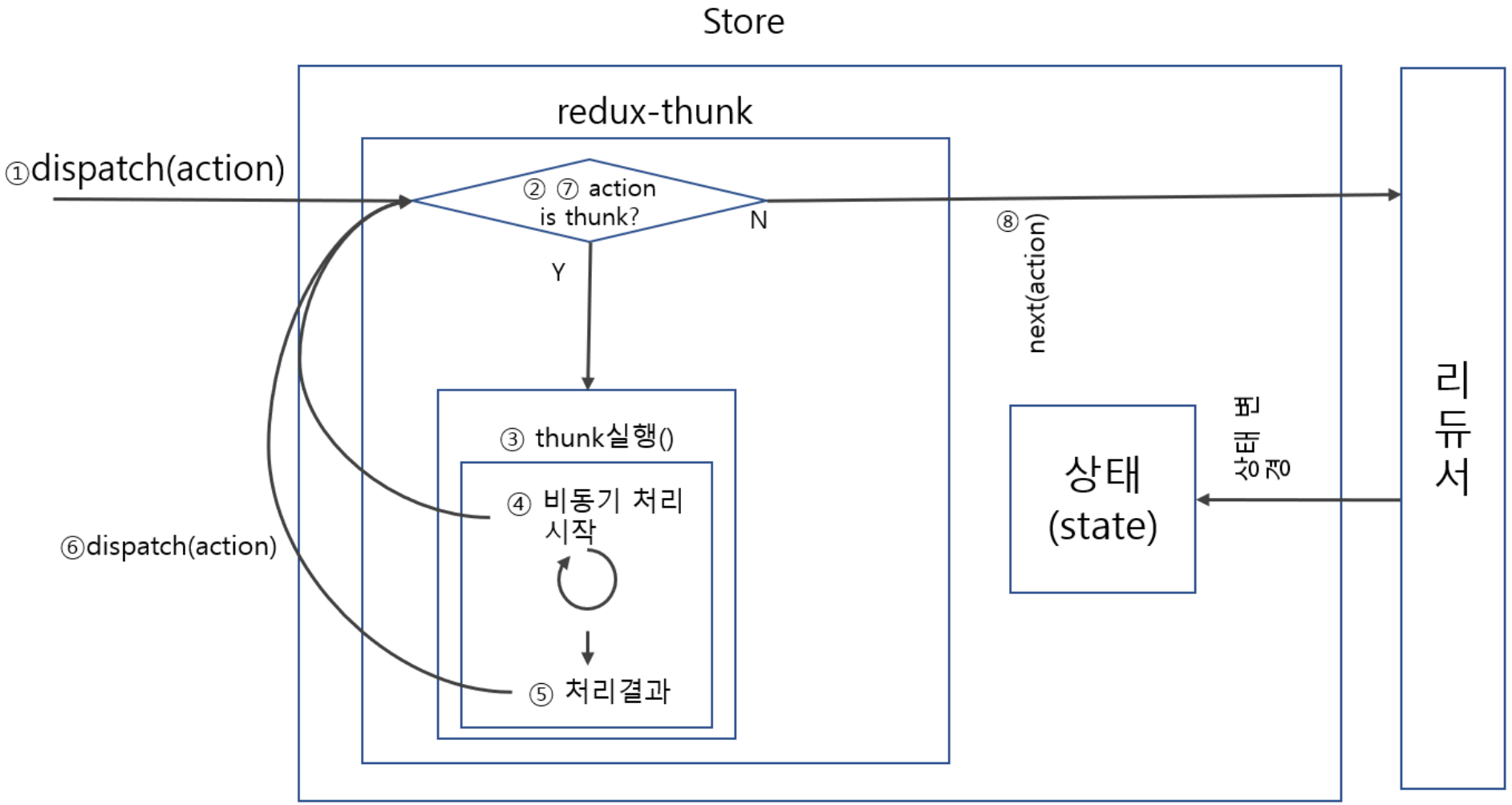
- 비동기 처리를 위한 redux용 미들웨어
- thunk
  - 컴퓨터 프로그램에서 다른 서브루틴 또는 함수로 연산 기능을 주입시킬 때 사용하는 함수
  - 지연된 실행을 위해 표현식으로 만든 함수
  - ActionCreator가 Action 메시지 대신에 thunk 함수를 리턴함
- 패키지 참조 : npm install redux-thunk
  - 리덕스 툴킷을 사용한다면 redux-thunk가 이미 포함되어 있으므로 추가설치할 필요 없음

❖redux-thunk의 적용 방법

- store 객체에서 미들웨어 등록

10. redux-thunk 미들웨어

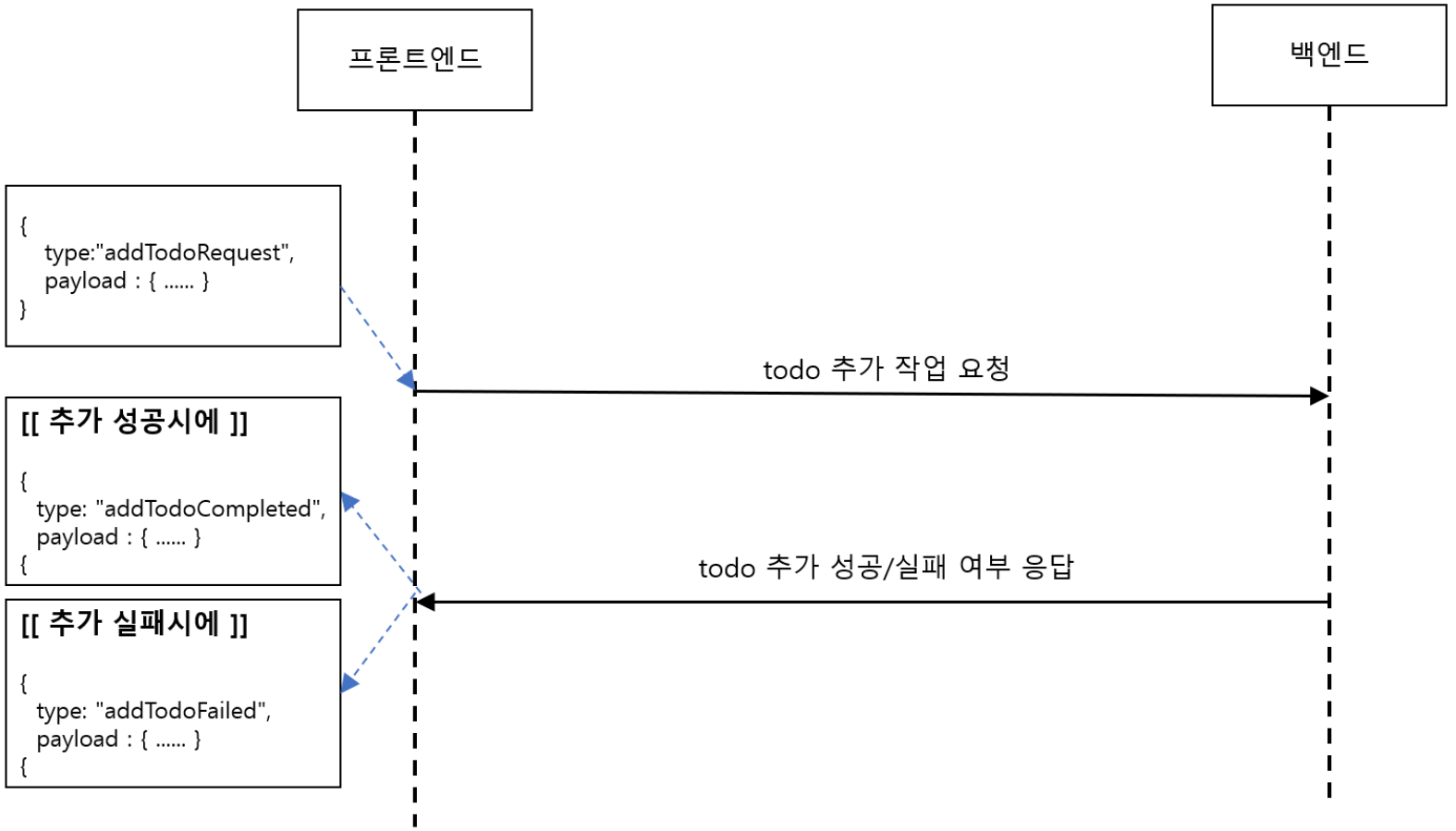
❖redux-thunk 미들웨어 아키텍처



10. redux-thunk 미들웨어

❖상태 변경이 필요한 시점

- 요청시점, 성공 응답, 실패 응답



* 시점별 액션명은 달라질 수 있음
-request : pending
-completed : fulfilled
-failed : rejected

10. redux-thunk 미들웨어

❖일반적인 thunk 함수 형태

```
//thunk 함수의 일반적인 패턴(async/await 버전)
async (dispatch: ThunkDispatch<TodoListStateType, argsType, Action>) => {
   try {
      //dispatch 함수를 이용해 작업의 시작 상태로 바꿈
      dispatch(ActionCreator.getTodosRequested());
      const response = await axios.get(url);
      //dispatch 함수를 이용해 작업 성공 상태와 함께 응답 데이터를 전달
      dispatch(ActionCreator.getTodosCompleted());
   } catch (error) {
      //dispatch 함수를 이용해 실패 시에는 에러 메시지를 담아 전달
      dispatch(ActionCreator.getTodosFailed());
   }
};
```

10. redux-thunk 미들웨어

❖순수하게 redux + redux-thunk로 개발하면...

- Action이 지나치게 복잡함.
  - 각 시점(pending, fulfilled, rejected)별 상태 변경을 액션 생성자가 필요함
  - 추가로 비동기 처리를 위해 thunk를 리턴하는 액션 생성자도 필요함
- 그렇기 때문에 RTK를 사용함

```
export type AppDispatch = ThunkDispatch<ContactStateType, undefined, UnknownAction> & Dispatch<UnknownAction>;

export const ContactActionCreator = {
  searchContactsPending : createAction("searchContactsPending"),
  searchContactsFulfilled : createAction("searchContactsFulfilled"),
  searchContactsRejected: createAction("searchContactsRejected"),
  asyncSearchContacts: (name) => {
    return async (dispatch:AppDispatchType, getState) => {
      let url = "https://localhost:3000/contacts_long/search/" + name;
      try {
        dispatch(ContactActionCreator.searchContactsPending({ name }));
        const response = await axios.get(url);
        dispatch(ContactActionCreator.searchContactsFulfilled({ contacts : response.data }));
      } catch (error) {
        dispatch(ContactActionCreator.searchContactsRejected({ status : error }));
      }
    };
  },
}
```

11. redux-thunk + RTK

❖RTK
- redux-thunk 패키지 이미 포함
- redux-thunk가 이미 defaultMiddleware로 등록되어 있음

❖상태 변경이 필요한 시점
- 작업 요청을 시작하는 시점 : asyncAction.pending
- 작업이 성공적으로 완료된 시점 : asyncAction.fulfilled
- 작업이 실패한 시점 : asyncAction.rejected

❖createAsyncThunk 툴킷 함수
- 요청 시작, 요청 완료 시점에 직접 dispatch 하지 않아도 됨.
- 내부적으로 ActionType과 액션 생성자를 만들어냄
  - 액션명을 "searchPerson"으로 지정했다면...

시점	액션명	액션 생성자 함수
비동기 작업 시작	searchPerson/pending	asyncAction.pending
비동기 작업 완료	searchPerson/fulfilled	asyncAction.fulfilled
비동기 작업 실패	searchPerson/rejected	asyncAction.rejected

11. redux-thunk + RTK

❖createAsyncThunk 함수 형태

```
createAsyncThunk<
   FulfilledResponseType,
   ArgumentType,
   ThunkAPIFieldType
> ( actionName, payloadCreater )
```

- 제네릭 타입
  - FulfilledResponseType
    - 비동기 요청 결과 성공시에 응답받는 데이터의 타입
  - ArgumentType
    - payloadCreator 함수의 첫번째 인자로 전달할 타입
    - 비동기 요청시에 전달하는 아규먼트 타입
    - 단일 값이므로 만일 여러개를 전달한다면 객체 타입으로 지정해야 함
  - ThunkAPIFieldType
    - 선택적, payloadCreater 함수의 두번째 인자인 thunkAPI 인자의 각 속성에 지정할 타입
    - dispatch?, state?, rejectValue?, extra?

11. redux-thunk + RTK

- 사용 인자
  - 첫번째 인자
    - 액션명을 문자열로 지정함
  - 두번째 인자
    - payloadCreator라는 비동기 처리 수행 함수
    - 요청/응답 시점별로 dispatch(action)하지 않아도 됨 --〉 자동으로 dispatch()를 수행함
    - 자동으로 dispatch하는 경우 pending, rejected 시점에는 action이 전달되지 않음

❖payloadCreater 함수의 형태

```
async ({ name }, thunkAPI) => {
  const url = "http://localhost:3000/contacts_long/search/" + name;
  const response = await axios.get<ContactItemType[]>(url);
  return { contacts: response.data };
}
```

- payloadCreater는 Promise 기반
  - async/await으로 작성하거나 Promise를 리턴하도록 작성되어야 함

11. redux-thunk + RTK

❖만일 각 요청, 응답 시점에 Action Payload를 전달해야 한다면?

- thunkAPI 인자를 사용함
  - thunkAPI.dispatch()
  - thunkAPI.getState()
  - thunkAPI.rejectWithValue(value, [meta])
  - thunkAPI.fulfillWithValue(value, meta)

- 요청 시점에 추가적인 액션을 전달하고 싶다면?
  - thunkAPI.dispatch({ type: "additionalAction", payload: { message: "추가적인 액션" } });
- 요청에 대한 응답 실패 시점에 에러메시지를 전달하고 싶다면?
  - return thunkAPI.rejectWithValue({ message: (err as unknown as Error).message });

11. redux-thunk + RTK

❖createAsyncThunk 함수의 제네릭 타입

- 코드 예시

```
const actionSearchContacts = createAsyncThunk<
  { contacts: ContactItemType[] },
  { name: string },
  {
    dispatch: AppDispatch,
    state: ContactStateType,
    rejectValue: ThunkErrorType,
  }
>("searchContacts", async ({ name }, thunkAPI) => {
    try {
      thunkAPI.dispatch({ type: "additionalAction", payload: { message: "추가적인 액션" } });
      const url = "https://contactsvc.bmaster.kro.kr/contacts_long/search/" + name;
      const response = await axios.get(url);
      return { contacts : response.data };
    } catch(err) {
      return thunkAPI.rejectWithValue({ message: (err as unknown as AxiosError).message })
    }
  }
);
```

11. redux-thunk + RTK

❖예제 프로젝트 생성

- npm init vite contacts-app -- --template react-swc-ts
- cd contacts-app
- npm install axios redux react-redux @reduxjs/toolkit

❖VSCode로 프로젝트 오픈한 뒤 다음과 같이 파일 정리

- App.css, assets 폴더 삭제
- 다음 파일 생성
  - src/redux/ContactAction.ts
  - src/redux/ContactReducer.ts
  - src/redux/ContactStore.ts
- index.css 변경

```
body { margin:20px; }
```

11. redux-thunk + RTK

❖src/redux/ContactAction.ts 작성

```
import { Dispatch, ThunkDispatch, UnknownAction, createAsyncThunk } from "@reduxjs/toolkit";
import axios from "axios";

export type AppDispatch = ThunkDispatch<ContactStateType, undefined, UnknownAction> &
Dispatch<UnknownAction>;

//백엔드에서 타입이 결정되고 상태의 contacts 속성 초기화가 되지 않았기 때문에 Utility Type을 이용할 수 없음
//백엔드 API의 응답결과를 확인하고 직접 타입을 선언함
export type ContactItemType = {
  no: string;
  name: string;
  tel: string;
  address: string;
  photo: string;
};
export type ContactStateType = {
  contacts: ContactItemType[];
  isLoading: boolean;
  status: string;
};

type ThunkErrorType = {
  message: string;
};
```

11. redux-thunk + RTK

❖src/redux/ContactAction.ts 작성

```
export const searchContactsAsync = createAsyncThunk<
  { contacts: ContactItemType[] },
  { name: string },
  {
    dispatch: AppDispatch;
    state: ContactStateType;
    rejectValue: ThunkErrorType;
  }
>("searchContacts", async ({ name }, thunkAPI) => {
  try {
    const url = "http://localhost:3000/contacts_long/search/" + name;
    const response = await axios.get(url);
    return { contacts: response.data };
  } catch (err) {
    //ⓐ rejectWithValue로 리턴값은 값은 리듀서에 의해 rejected 시점에 실행됨
    return thunkAPI.rejectWithValue({ message: (err as unknown as Error).message });
  }
});
```

11. redux-thunk + RTK

❖src/redux/ContactReducer.ts 작성

```
import { createReducer } from "@reduxjs/toolkit";
import { ContactStateType, searchContactsAsync } from "./ContactAction";

const initialState: ContactStateType = { contacts: [], isLoading: false, status: "" };

const ContactReducer = createReducer(initialState, (builder) => {
  builder
    .addCase(searchContactsAsync.pending, (state, action) => {
      state.isLoading = true;
      state.status = action.meta.arg.name + " 포함 이름으로 조회중";
    })
    .addCase(searchContactsAsync.fulfilled, (state, action) => {
      state.contacts = action.payload.contacts;
      state.isLoading = false;
      state.status = "조회 완료";
    })
    .addCase(searchContactsAsync.rejected, (state, action) => {
      state.contacts = [];
      state.isLoading = false;
      state.status = "조회 실패 : " + action.payload && action.payload?.message ? action.payload.message : "알 수 없는 오류";
    });
});
export default ContactReducer;
```

11. redux-thunk + RTK

❖src/redux/ContactStore.ts 작성

```
import { Middleware, configureStore } from "@reduxjs/toolkit";
import ContactReducer from "./ContactReducer";

const logger: Middleware = (store) => (next) => (action) => {
  console.log("action ", action);
  next(action);
  console.log("state: ", store.getState());
};

const ContactStore = configureStore({
  reducer: ContactReducer,
  middleware: (getDefaultMiddleware) => {
    return getDefaultMiddleware({ serializableCheck: false }).concat([logger]);
  },
});

export default ContactStore;
```

11. redux-thunk + RTK

❖src/App.tsx 작성

```
import { useState } from "react";
import { useDispatch, useSelector } from "react-redux";
import { AppDispatch, ContactItemType, ContactStateType, searchContactsAsync } from "./redux/ContactAction";

type PropsType = {
  contacts: ContactItemType[];
  isLoading: boolean;
  status: string;
  searchContacts: (name: string) => void;
};
const App = ({ contacts, isLoading, status, searchContacts }: PropsType) => {  ……(생략)};

const AppContainer = () => {
  const dispatch = useDispatch<AppDispatch>();
  const propsObject = {
    isLoading: useSelector((state: ContactStateType) => state.isLoading),
    status: useSelector((state: ContactStateType) => state.status),
    contacts: useSelector((state: ContactStateType) => state.contacts),
    searchContacts: (name: string) => dispatch(searchContactsAsync({ name })),
  };
  return <App {...propsObject} />;
};

export default AppContainer;
```

11. redux-thunk + RTK

❖src/main.tsx 변경

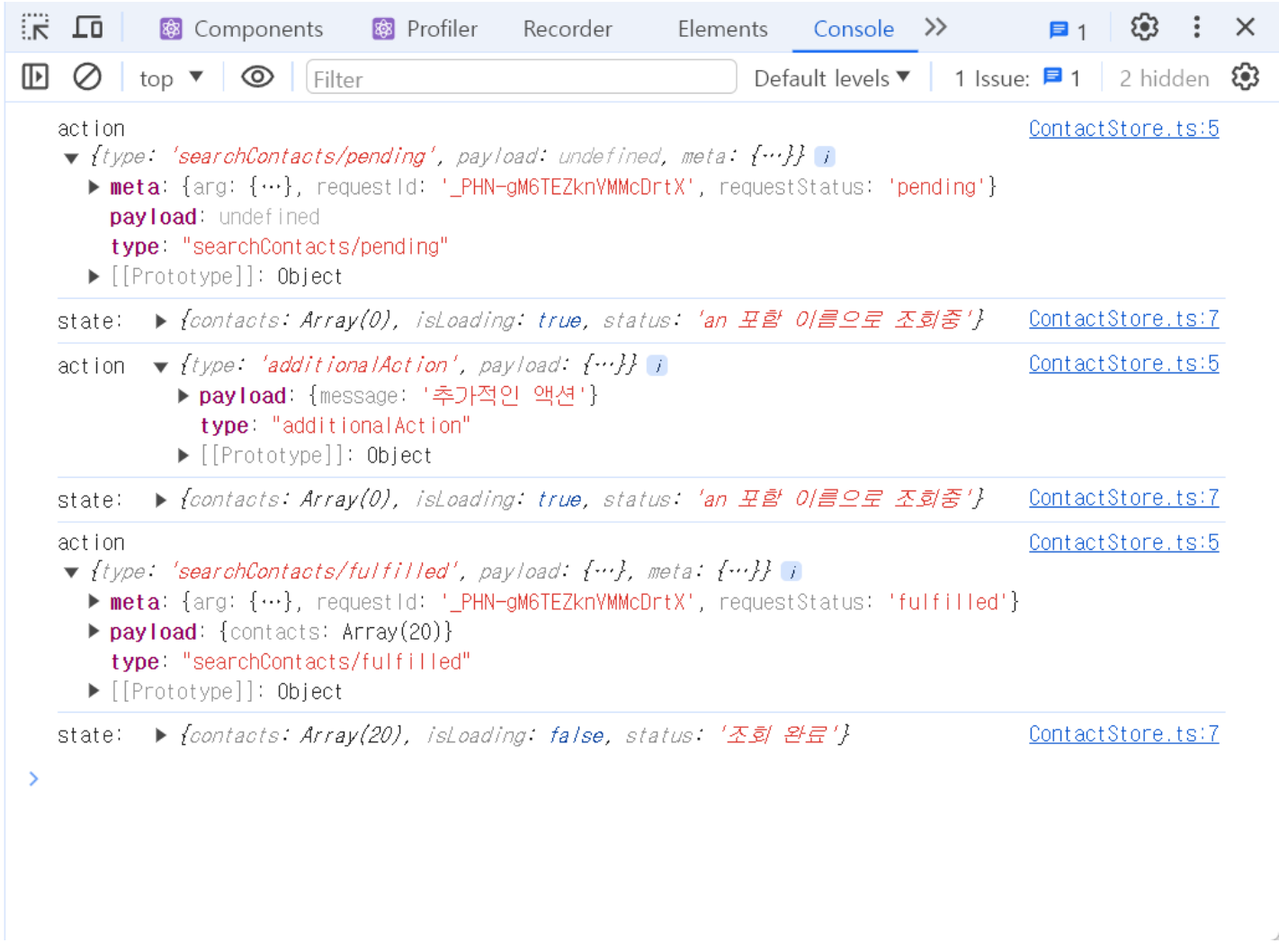
```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.tsx'
import './index.css'
import { Provider } from 'react-redux'
import ContactStore from './redux/ContactStore.ts'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
    <Provider store={ContactStore}>
      <App />
    </Provider>
  </React.StrictMode>,
)
```

11. redux-thunk + RTK

❖실행 결과1 : 성공했을 때

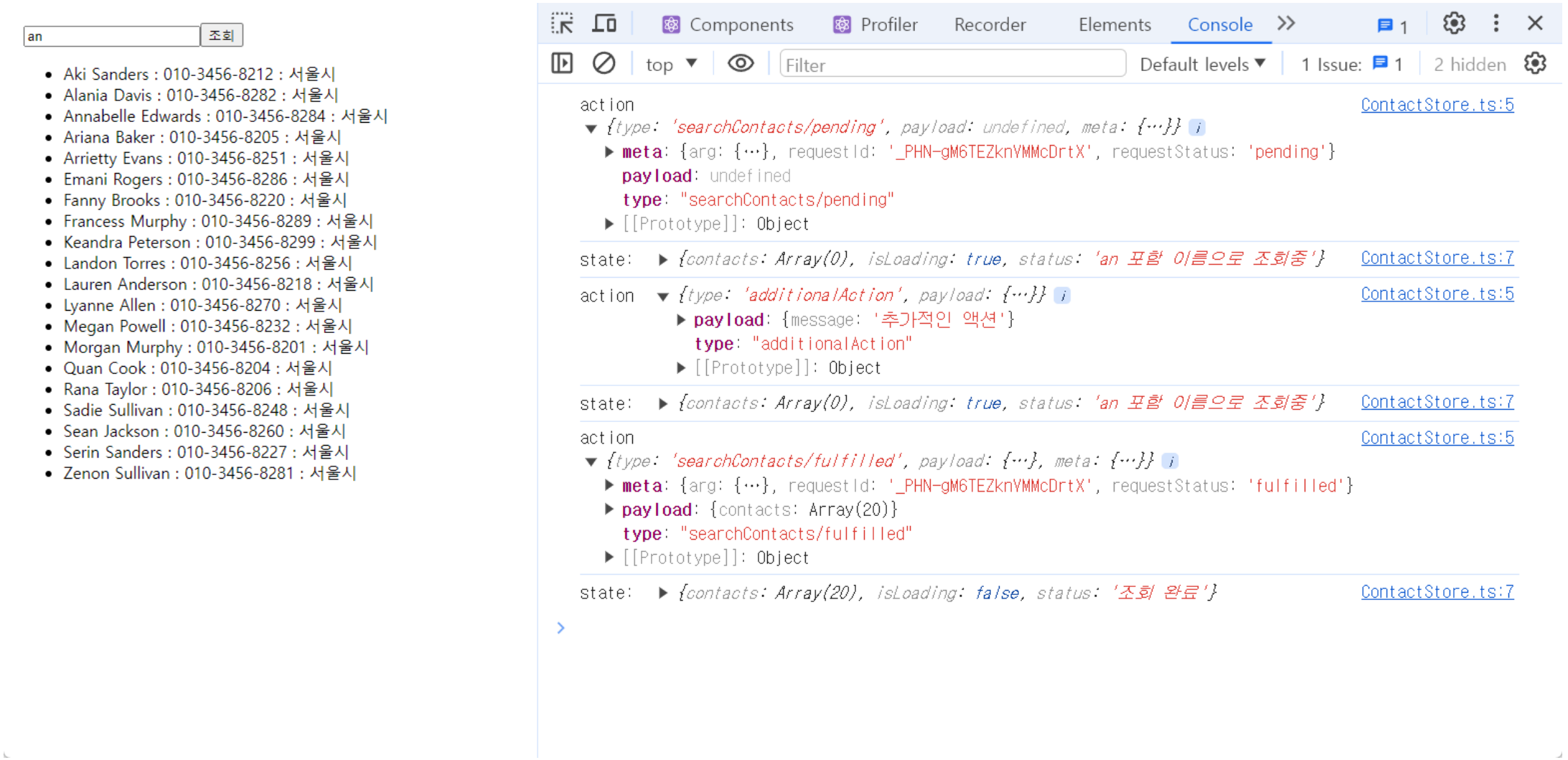
an 조회

- Aki Sanders : 010-3456-8212 : 서울시
- Alania Davis : 010-3456-8282 : 서울시
- Annabelle Edwards : 010-3456-8284 : 서울시
- Ariana Baker : 010-3456-8205 : 서울시
- Arrietty Evans : 010-3456-8251 : 서울시
- Emani Rogers : 010-3456-8286 : 서울시
- Fanny Brooks : 010-3456-8220 : 서울시
- Francess Murphy : 010-3456-8289 : 서울시
- Keandra Peterson : 010-3456-8299 : 서울시
- Landon Torres : 010-3456-8256 : 서울시
- Lauren Anderson : 010-3456-8218 : 서울시
- Lyanne Allen : 010-3456-8270 : 서울시
- Megan Powell : 010-3456-8232 : 서울시
- Morgan Murphy : 010-3456-8201 : 서울시
- Quan Cook : 010-3456-8204 : 서울시
- Rana Taylor : 010-3456-8206 : 서울시
- Sadie Sullivan : 010-3456-8248 : 서울시
- Sean Jackson : 010-3456-8260 : 서울시
- Serin Sanders : 010-3456-8227 : 서울시
- Zenon Sullivan : 010-3456-8281 : 서울시



11. redux-thunk + RTK

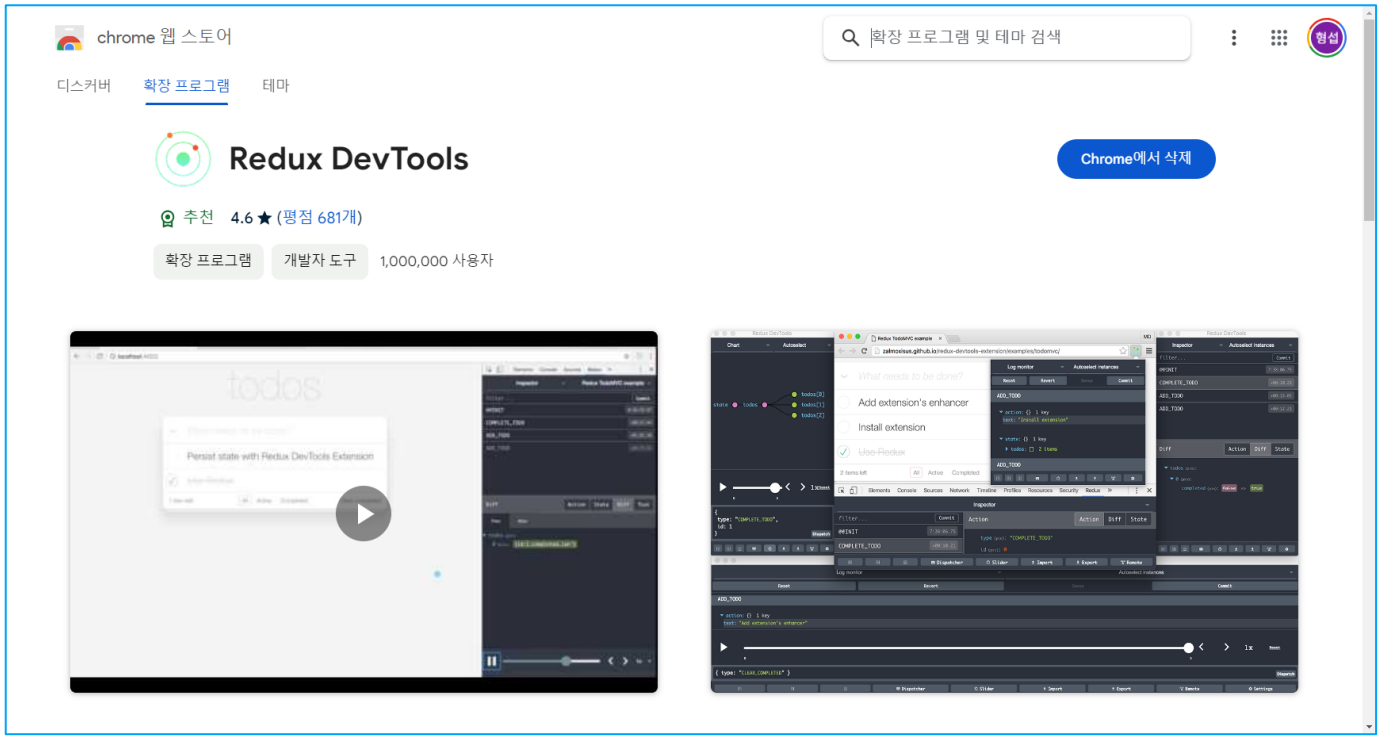
❖실행 결과 2 : 실패했을 때 영문 한글자만 입력하고 조회시도



12. Redux Devtools

❖Redux Devtools

- Redux를 이용한 앱을 개발할 때 개발을 강력하게 지원하는 개발 패키지 도구
  - Redux의 상태와 액션 정보를 시각화하며, 상태 변경을 추적할 수 있도록 함.
- 크롬 확장 프로그램 설치
  - Redux Devtools로 구글링하여 크롬 확장 프로그램 설치



12. Redux Devtools

- Redux Dev tools는 미들웨어로 작성되었음
  - Store에 미들웨어 설정만으로 적용 끝
    - @reduxjs/toolkit에서는 미들웨어의 등록이 이미 되어 있음
  - Redux의 불변성 --〉 시간 여행 디버깅을 가능하게 함 --〉 Redux Devtools
  - 개발환경일 때만 사용하도록 하기 위한 설정이 필요함
- todolist-app-router 예제에 적용

```
//logger를 사용하지 않도록 설정
const AppStore = configureStore({
  reducer: RootReducer,
  middleware: (getDefaultMiddleware) => {
    return getDefaultMiddleware({ serializableCheck: false });
  },
  devTools: process.env.NODE_ENV !== "production"
});

export default AppStore;
```

12. Redux Devtools

❖ Demo

- 시간 여행 디버깅
- 상태 변경 추적, Diff, Chart