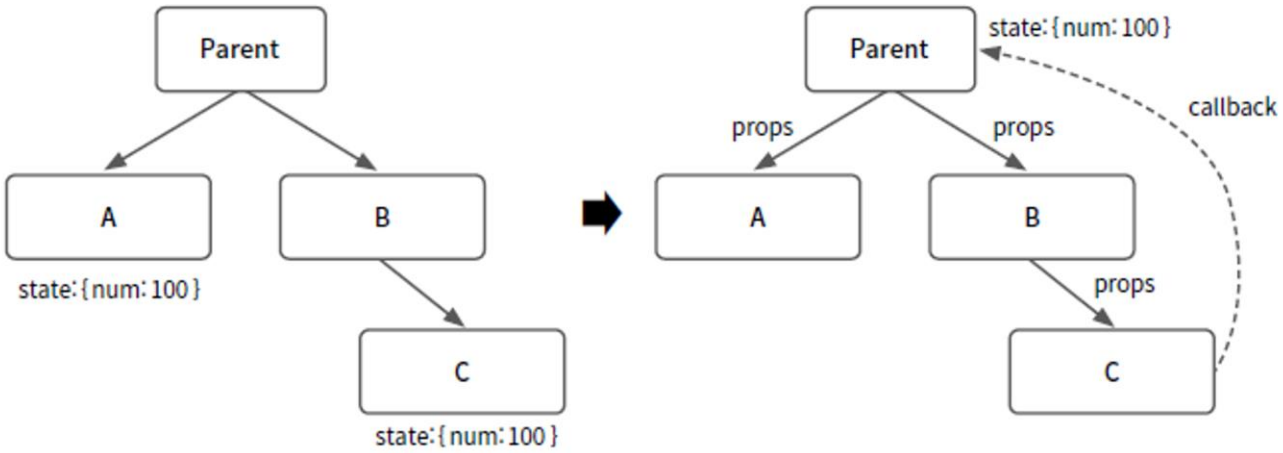
- Redux 아키텍처 정리
- 기존 예제 리팩토링

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 나머지는 필요할 때 문서 정독 후 그냥 사용하면 됨

3. Redux 소개

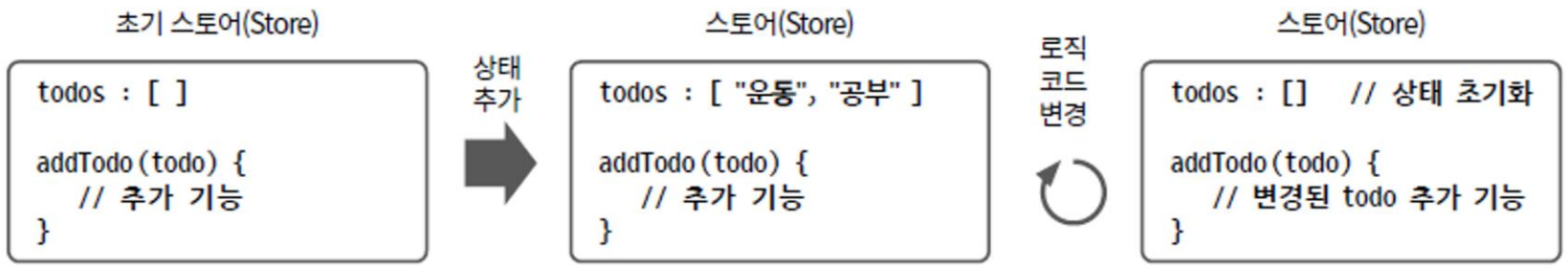
❖Redux?

- 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)

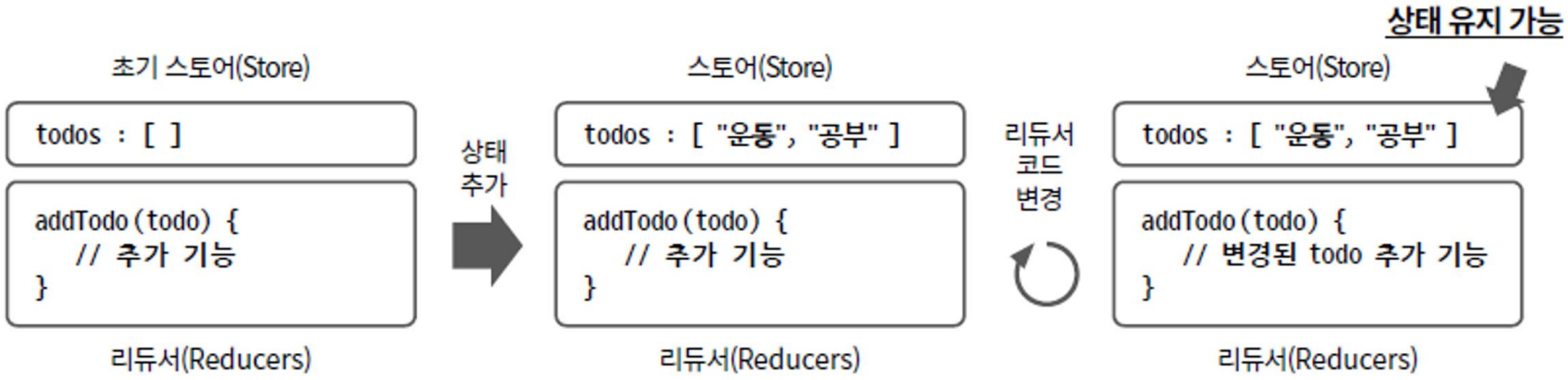
3. Redux 소개

❖Redux 특징 1

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



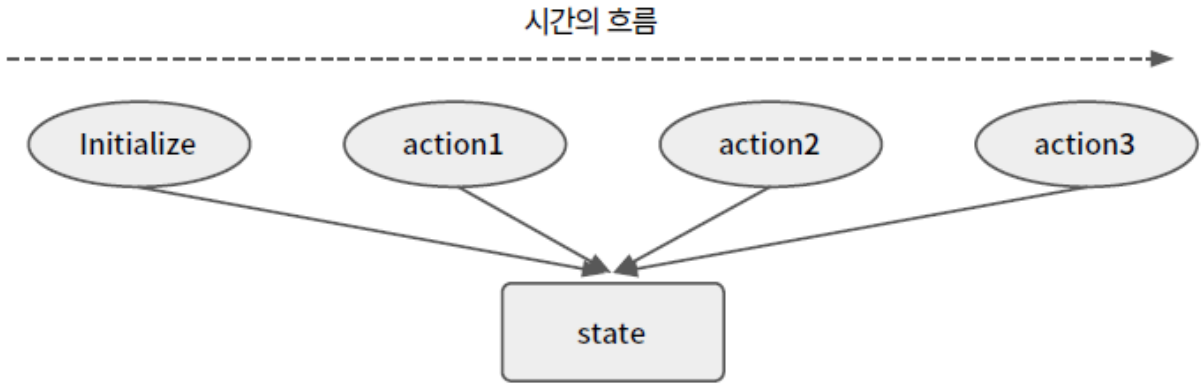
- Redux : 상태와 상태 변경 기능(Reducer)을 분리



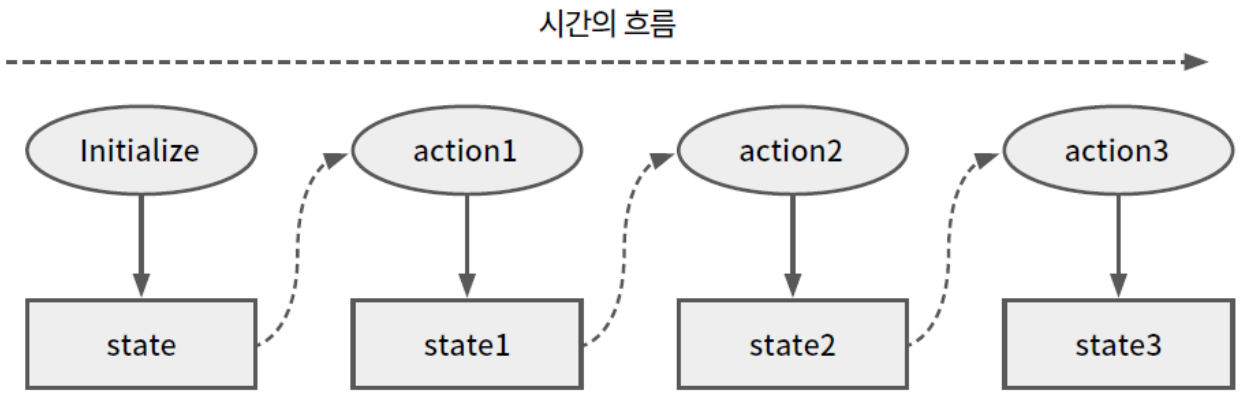
3. Redux 소개

❖Redux 특징 2

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

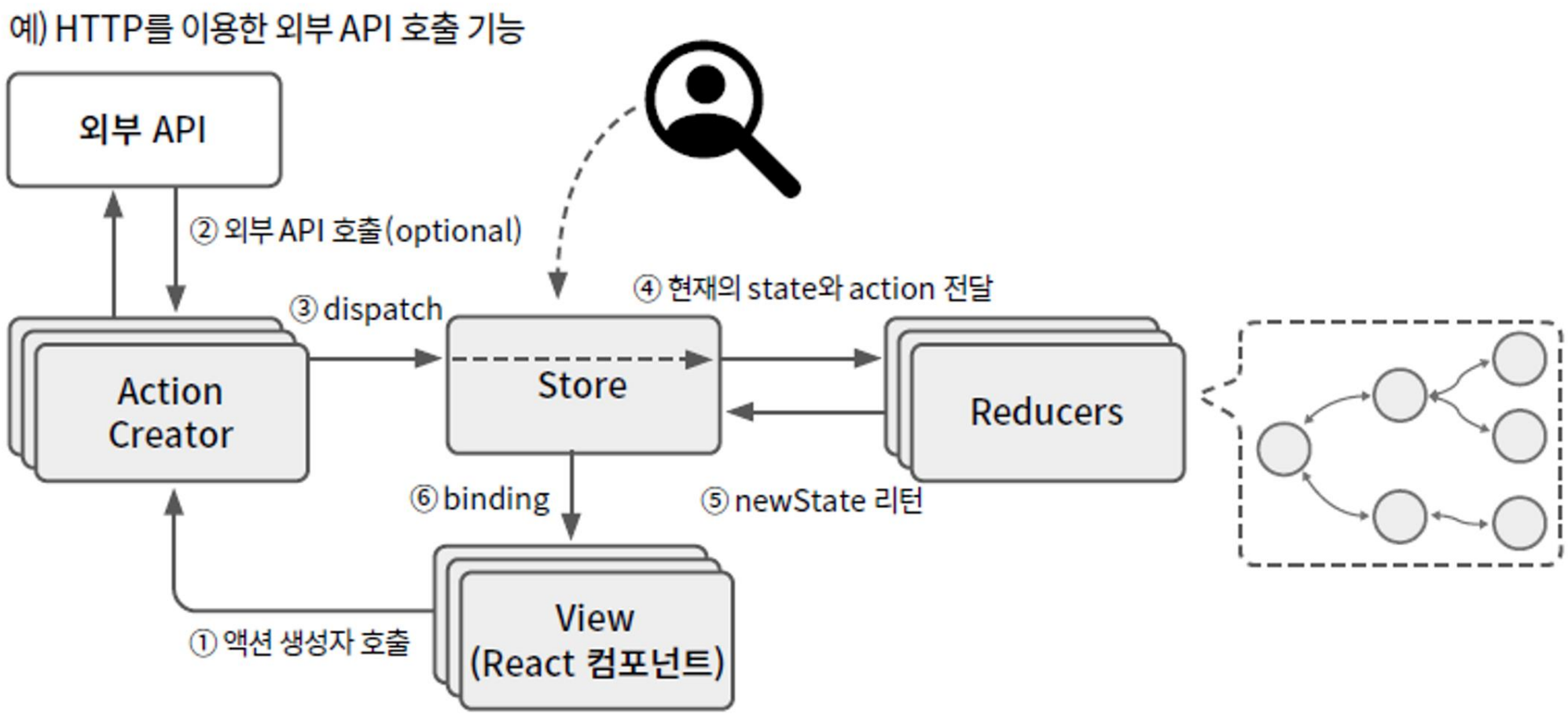


- Redux의 상태 변경 : 불변성 필수 --〉 상태 변경 추적 --〉 시간여행 디버깅!!



3. Redux 소개

❖ Redux 아키텍처



3. Redux 소개

❖Redux 아키텍처에서의 처리 흐름

①, ② 스토어의 상태와 액션 생성자를 이용한 디스패치 기능의 함수가 주입된 뷰 컴포넌트(리액트 컴포넌트)에서 이벤트가 발생합니다.

③ 뷰 컴포넌트의 이벤트 핸들러는 속성으로 주입된 함수가 호출되며, 이때 action(객체, 메시지)을 만들어서 스토어로 전달(dispatch)합니다. 액션의 형태는 플럭스 아키텍처에서 사용하던 것과 유사합니다.

* 예시: { type: "addTodo", payload: { todo: "강변 달리기", desc: "출근 전 30분 뛰기" } }

④ 스토어는 전달받은 액션과 자신의 상태를 리듀서 함수의 인자로 전달합니다. 리듀서는 다음과 같은 형식의 함수입니다.

* 리듀서 형식: (state, action) => { .... }

⑤ 리듀서는 인자로 전달받은 상태(state)는 변경하지 않고, action을 이용해 새로운 상태를 만든 뒤 리턴합니다.

⑥ 스토어는 리듀서가 리턴한 상태를 새로운 상태로 설정합니다. 스토어의 새로운 상태는 뷰 컴포넌트에 연결(binding)되어 있으므로 화면이 새롭게 렌더링됩니다.

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)

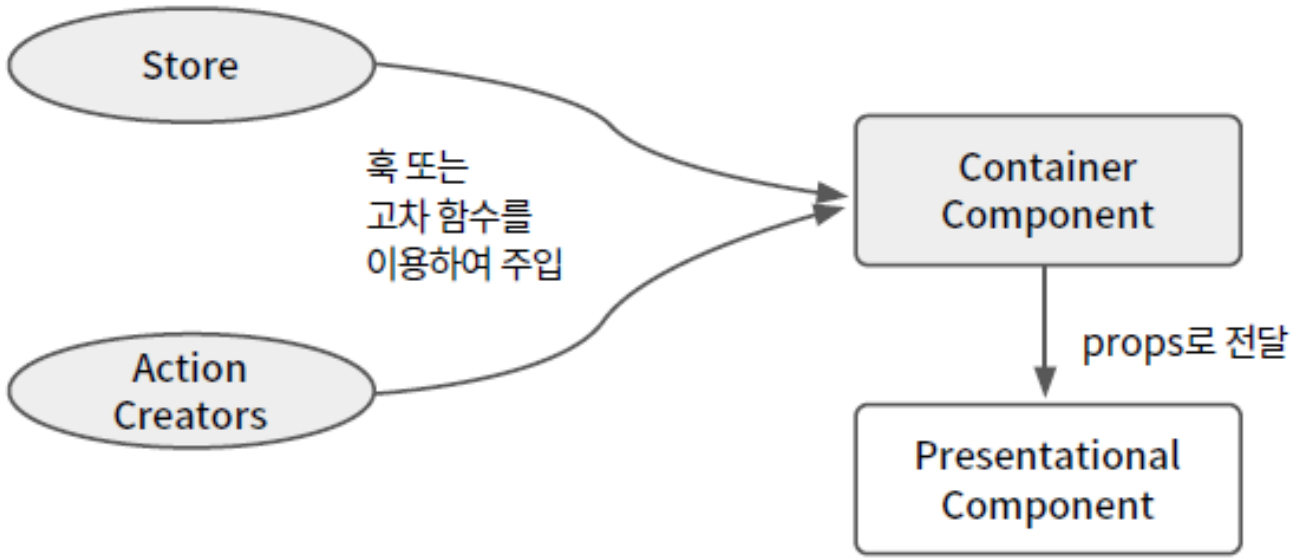
```
//자바스크립트 배열의 reduce 메서드가 사용하는 리듀서 함수
(sum, num) => {
  return sum + num;       //새로운 합계값 리턴
}

//리덕스의 리듀서 함수
(state, action) => {
  ......
  return newState;        //새로운 상태를 만들어서 리턴
}
```

- 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

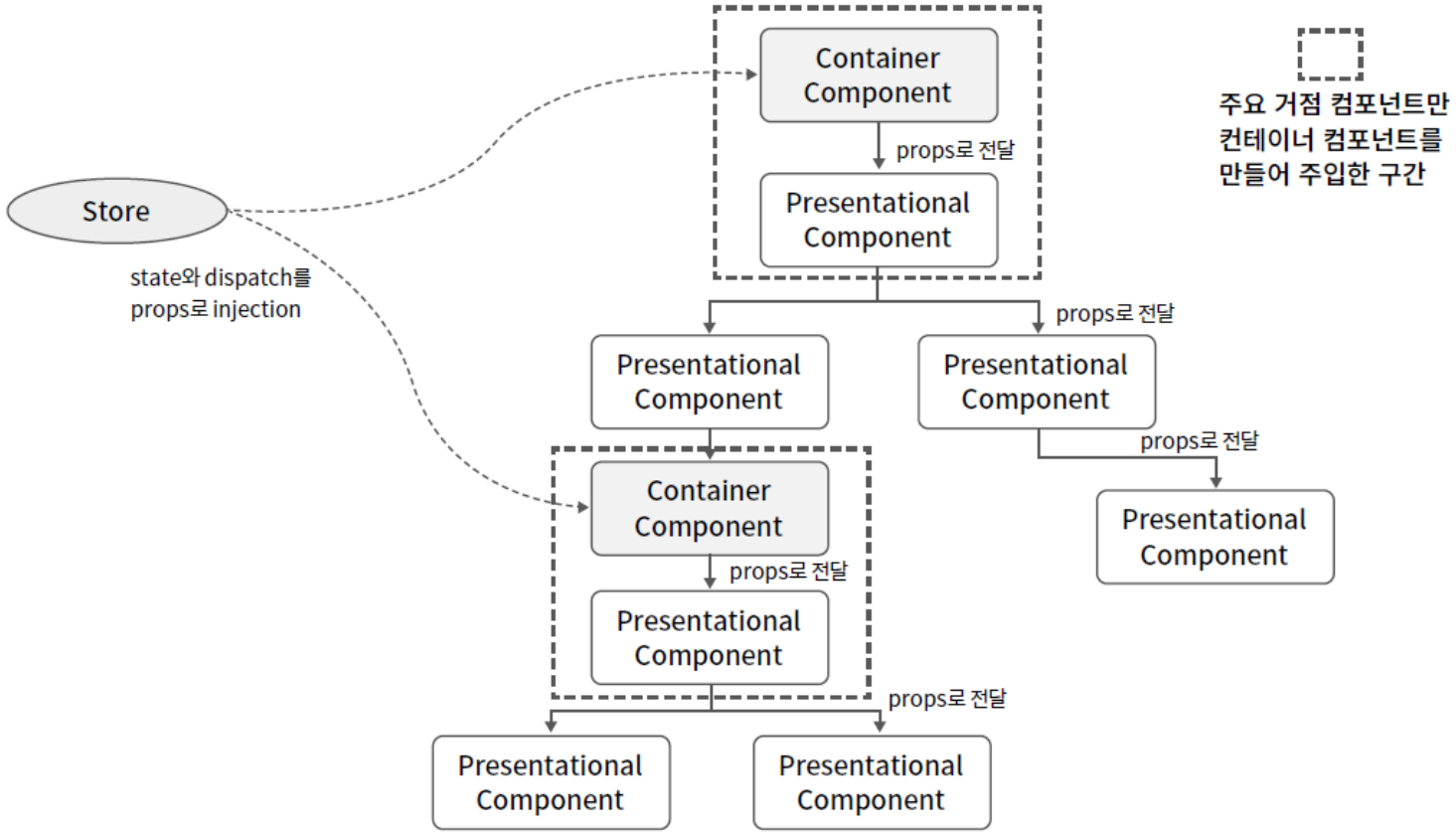
❖ Redux 컨테이너 컴포넌트

- 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

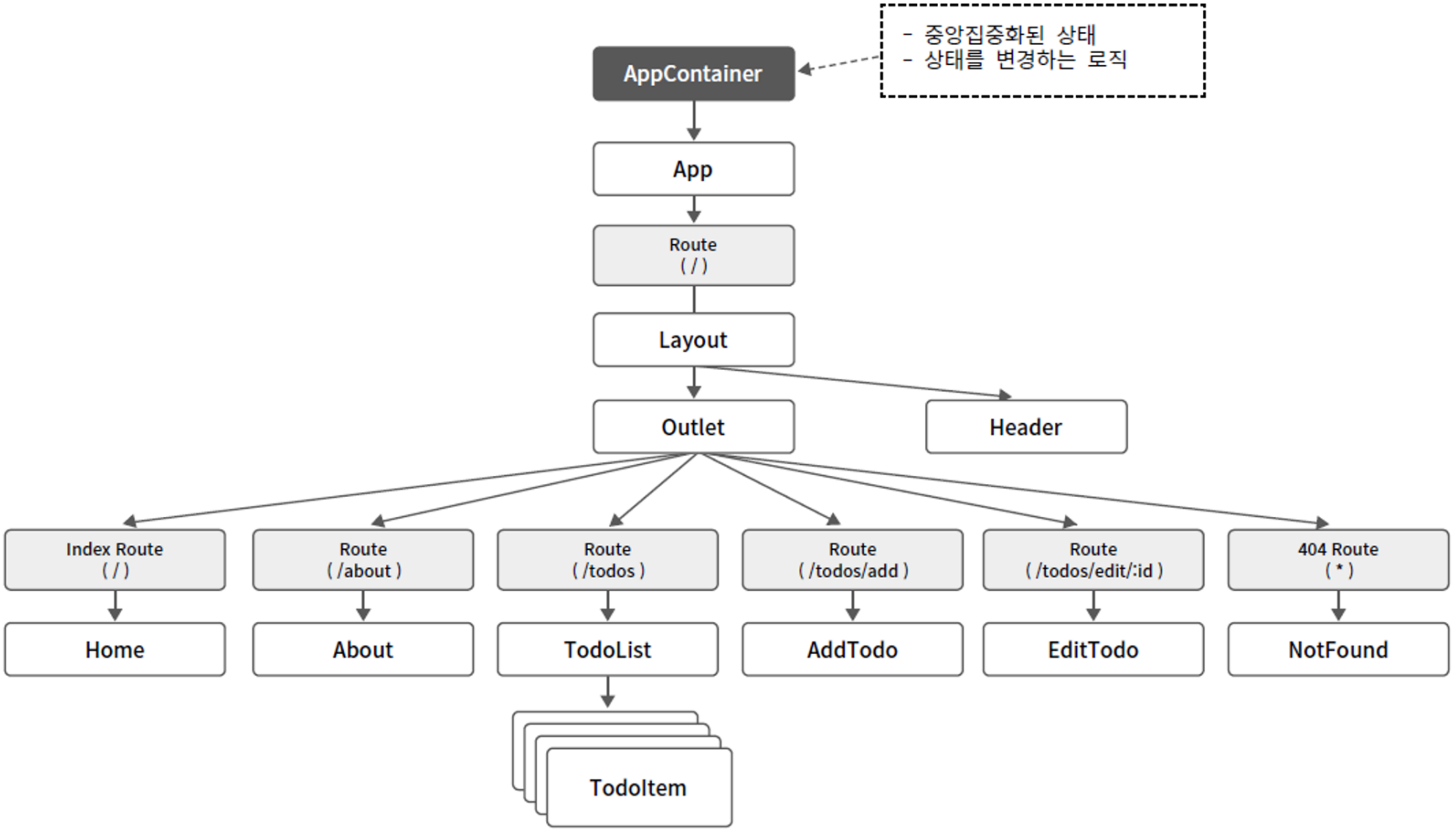
❖ react-redux가 제공하는 훅

- 훅을 사용하는 것이 더 직관적으로 느껴짐

- useStore( ): 리덕스 스토어 객체를 리턴합니다. 스토어의 상태를 읽어내려면 이 객체의 getState( ) 함수를 이용합니다.
- useDispatch( ): 스토어의 dispatch( ) 함수를 리턴합니다. 리턴받은 함수를 이용해 액션을 스토어로 전달할 수 있습니다.
- useSelector( ): 리덕스 스토어의 특정 상태를 선택하여 리턴합니다.

4. 미리 제공되는 예제

❖제공 예제 아키텍처



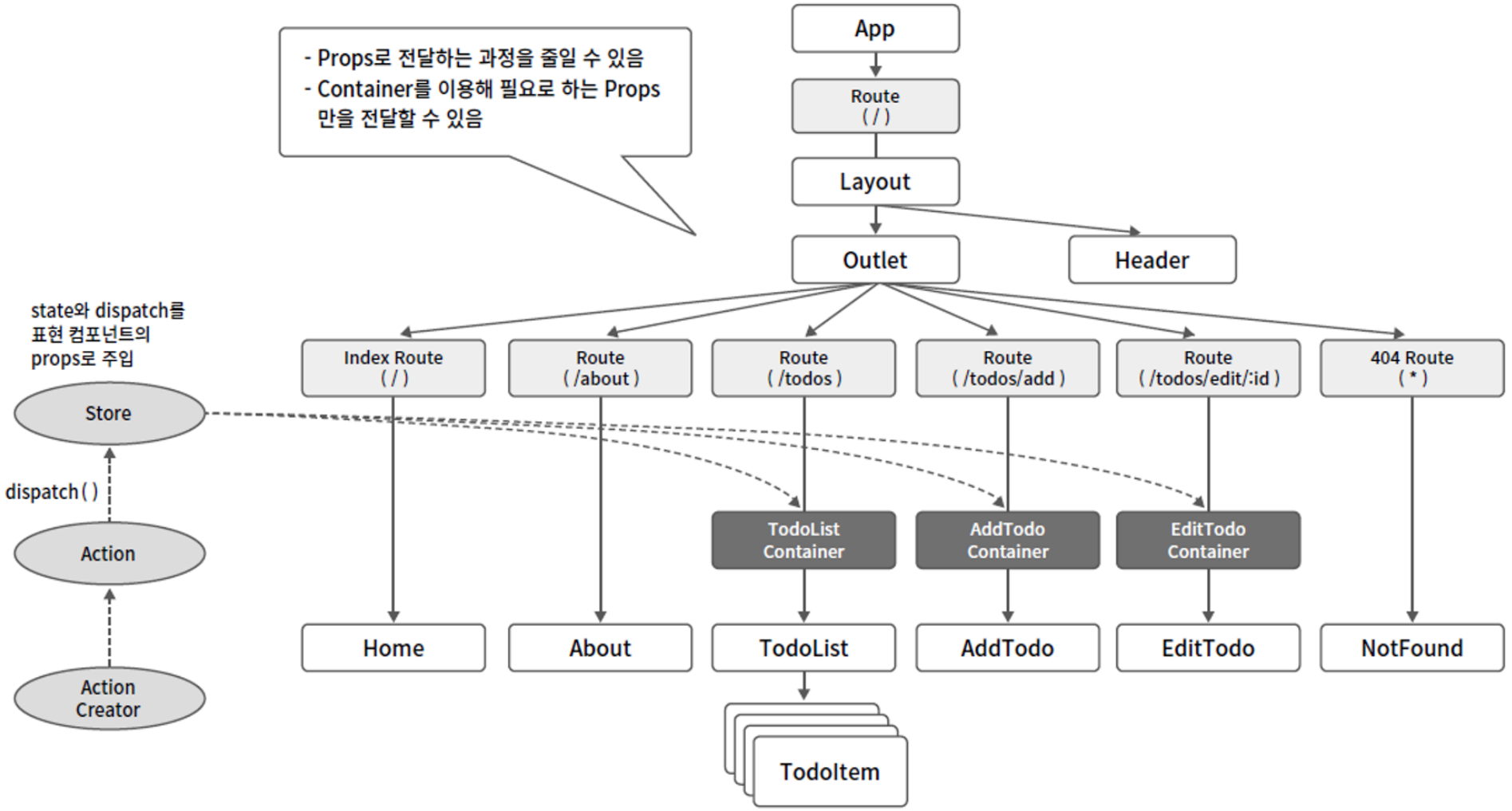
4. 미리 제공되는 예제

❖ 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
  - Redux Toolit(X) VS Redux Toolkit(O)

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

todolist-app-router의 상태

이 애플리케이션에서 리덕스를 이용해 전역 수준에서 관리할 상태는 todoList 데이터입니다. 기존 컴포넌트에서 사용하던 모든 상태를 전역 수준에서 관리할 필요는 없습니다. 다음의 경우에 해당한다면 로컬 컴포넌트의 상태를 그대로 이용하세요.

* 특정 컴포넌트에서만 사용되는 상태인 경우
* 상태 변경을 추적할 만큼 중요한 상태가 아닌 경우
* 컴포넌트의 생명주기가 바뀌더라도(예를 들어 다른 화면에 이동한 후 다시 돌아오더라도) 상태가 유지될 필요가 없는 경우

사용할 액션(action)

오로지 상태가 바뀌는 작업으로 한정하여 정의합니다. 이 애플리케이션에서는 네 가지의 상태 변경 작업이 있습니다. 그리고 각각에 대해 다음 표를 참고하여 액션의 형식도 함께 지정합니다. 액션의 형식 중 payload 부분은 상태를 변경할 때 어떤 값이 필요한지를 생각해보면 됩니다.

액션명	액션 형식
addTodo	{ type: "addTodo", payload: { todo: string, desc: string } }
deleteTodo	{ type: "deleteTodo", payload: { id: number } }
toggleDone	{ type: "toggleDone", payload: { id: number } }
updateTodo	{ type: "updateTodo", payload: { id: number, todo: string, desc: string, done: boolean } }

5. 1단계 - Redux 적용

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

- Action 메시지 객체를 생성하여 리턴함

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

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

5. 1단계 - Redux 적용

❖리듀서 작성 : src/redux/TodoReducer.js

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

const initialState = {
  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 },
  ],
};

export const TodoReducer = (state=initialState, action) => {
  let index;
  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:
      index = state.todoList.findIndex((item) => item.id === action.payload.id);
      return produce(state, (draft) => {
        draft.todoList.splice(index, 1);
      });
    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;
  }
};
```

5. 1단계 - Redux 적용

❖리듀서 gotjf

- 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=initialState, action) => {···
};
```

  - 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.js

- 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

- 속성을 states, callbacks로 묶어서 전달하지 않고 필요한 것만을 전달하도록 코드 변경
  - 속성 : todoList, deleteTodo, toggleDone
- 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 필요한 것만 속성으로 전달

5. 1단계 - Redux 적용

❖TodoItem 컴포넌트 변경

- connect() 이용하여 컨테이너 컴포넌트를 작성하지 않음
- 전달받는 속성만 변경 --〉 callbacks 이용하지 않음
  - todoItem, deleteTodo, toggleDone

❖AddTodo 컴포넌트

- 속성 : addTodo 속성만 전달받도록 작성
- useDispatch() 훅을 이용해 액션을 전달하도록 컨테이너 작성

❖EditTodo 컴포넌트 변경

- 속성 변경 : updateTodo, todoList
- useDispatch() 훅을 이용해 updateTodo 액션 전달 메서드를 속성으로 전달하도록 컨테이너 작성

5. 1단계 - Redux 적용

❖ 예제 : App.jsx 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.jsx 변경

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

const TodoList = ({ todoList, deleteTodo, toggleDone }) => {
  let todoItems = todoList.map((item) => {
    return <TodoItem key={item.id} todoItem={item}
              deleteTodo={deleteTodo} toggleDone={toggleDone} />;
  });
  return (
    <>
      <div className="row">
        <div className="col p-3">
          <Link className="btn btn-primary" to="/todos/add">
            할일 추가
          </Link>
        </div>
      </div>
      <div className="row">
        <div className="col">
          <ul className="list-group">{todoItems}</ul>
        </div>
      </div>
    </>
  );
};
```

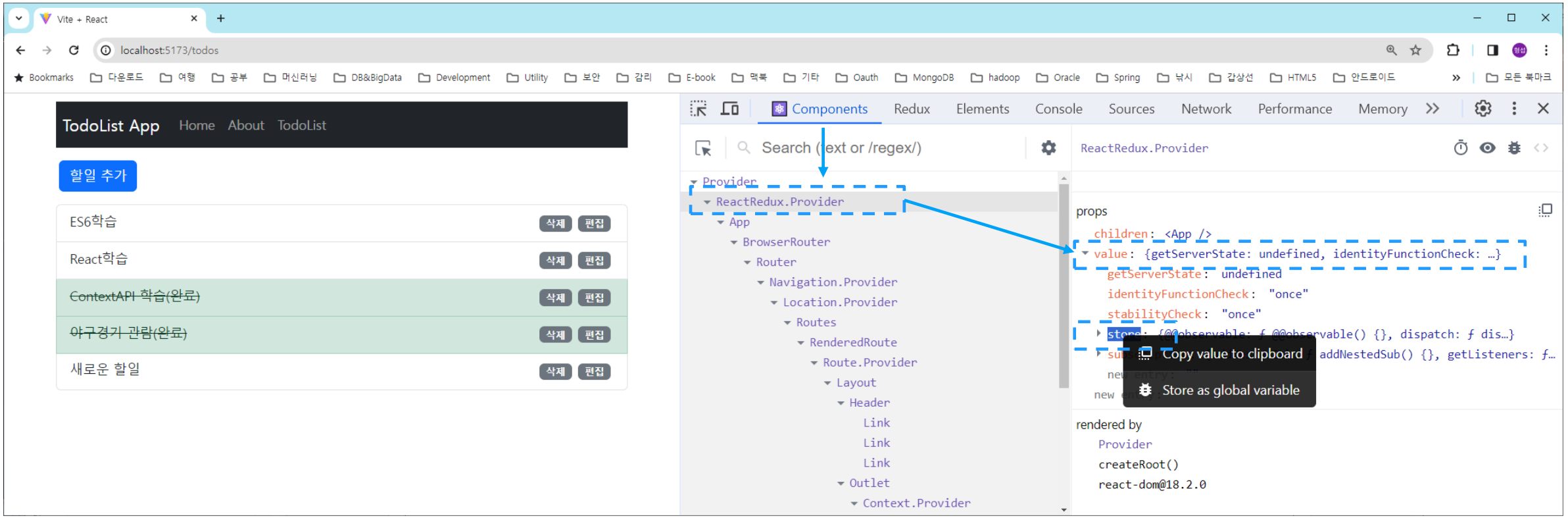
```
const TodoListContainer = ()=>{
  const dispatch = useDispatch();
  const todoList = useSelector((state)=>state.todoList);
  const toggleDone = (id) => dispatch(TodoActionCreator.toggleDone({id}))
  const deleteTodo = (id) => dispatch(TodoActionCreator.deleteTodo({id}))
  return <TodoList todoList={todoList}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

export { TodoList }
export default TodoListContainer;
```

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

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

//변경된 예제 : RTK 사용(O)
export const TodoActionCreator = {
  addTodo: createAction("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, TodoReducer.js 변경

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

export const TodoActionCreator = {
  addTodo: createAction("addTodo"),
  deleteTodo: createAction("deleteTodo"),
  toggleDone: createAction("toggleDone"),
  updateTodo: createAction("updateTodo"),
};
```

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

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

6. 2단계 - Redux Toolkit 적용

❖액션 생성자, 리듀서를 한번에 만들면 안될까?

- 그래서 등장한 것이 createSlice()

❖createSlice()

- 초기 상태, 리듀서 함수, 이름으로 구성된 객체(슬라이스)를 옵션 인자로 전달받아 리듀서와 상태에 해당하는 액션 생성자와 액션 유형을 자동으로 생성하는 고차함수
- slice : initialState + Reducer + Name + ActionType
- "todolist-app-router-2단계 완성-slice" 예제 검토할 것

❖너무 높은 추상화는 개발자의 시야를 흐리게 함

- 라이브러리를 사용은 하되 매몰되지 말자!!
- 버전 바뀌면 다시 배워야 함.
- 그래서 아키텍처를 이해하는 것이 중요함

7. 다중 리듀서

❖애플리케이션의 상태가 복잡해지면?

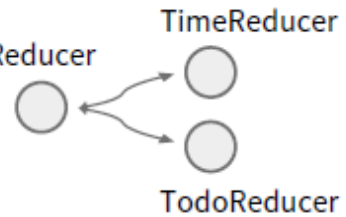
- 리듀서의 상태 변경 기능도 많아지고 복잡해짐
  - 하나의 리듀서로 처리 불가능
  - 따라서 여러 개의 리듀서(다중 리듀서)로 분리시켜야 함
- 다중 리듀서를 사용하려면...
  - 자식 리듀서들은 전체 상태 트리 중 특정 하위의 트리를 담당하기 때문에 상태 트리를 꼼꼼하게 설계해야 함

- 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 주입해야할 상태가 다름
  - const todoList = useSelector((state)=>**state.todos.todoList**);

7. 다중 리듀서

❖TimeActionCreator, TimeReducer 작성

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

export const TimeActionCreator = {
  changeTime : createAction("changeTime"),
};
```

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

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

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

7. 다중 리듀서

❖ AppStore 변경

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

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.js 추가

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

export default MyTime;
```

7. 다중 리듀서

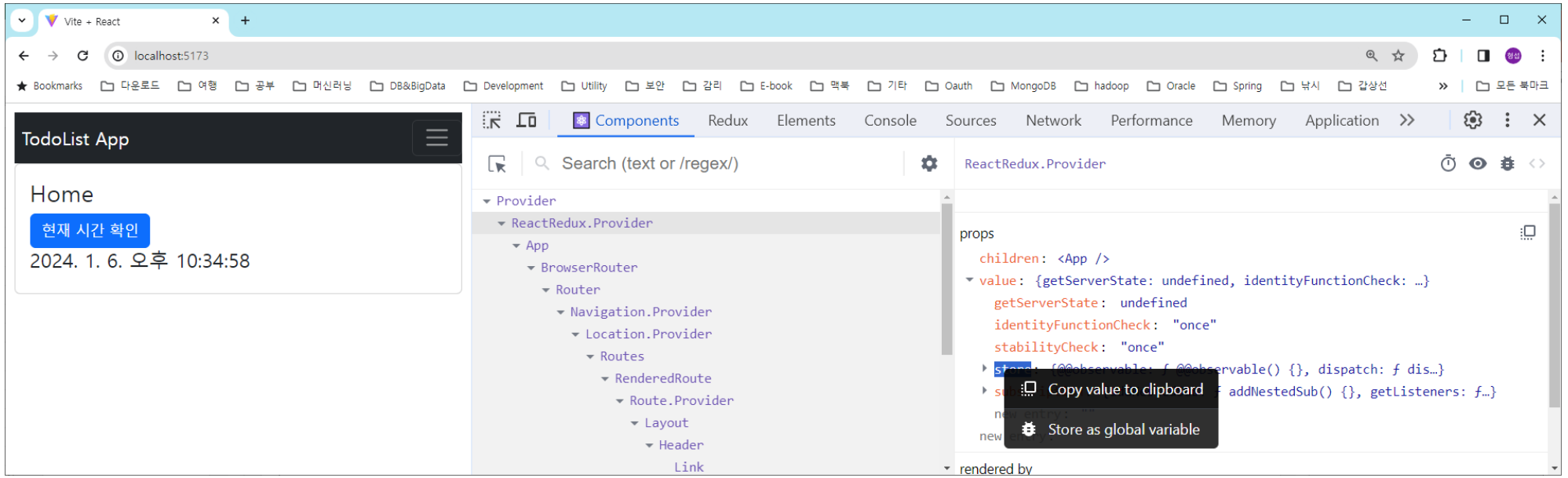
❖Home 컴포넌트 변경

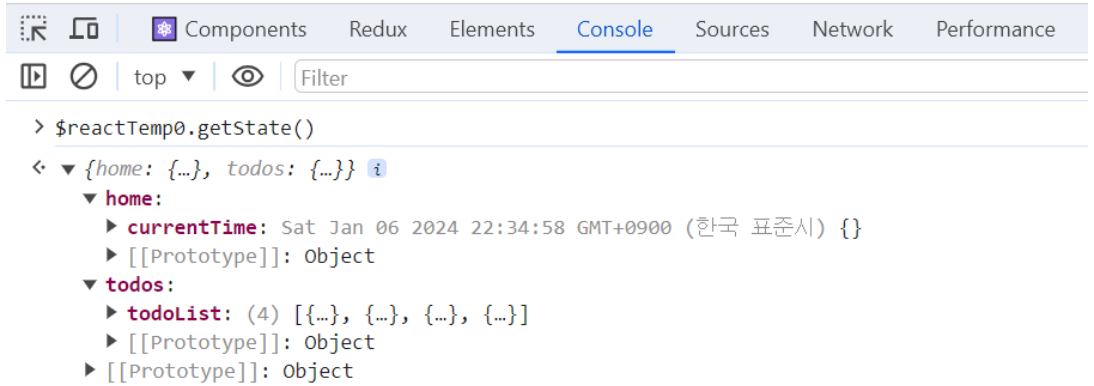
```
import MyTime from "./MyTime";
import { TimeActionCreator } from "../redux/TimeActionCreator";
import { useDispatch, useSelector } from "react-redux";

const Home = ({ currentTime, changeTime }) => {
  return (
    <div className="card card-body">
      <h2>Home</h2>
      <MyTime currentTime={currentTime} changeTime={changeTime} />
    </div>
  );
};

export const HomeContainer = () => {
  const dispatch = useDispatch();
  const currentTime = useSelector((state)=>state.home.currentTime);
  const changeTime = (currentTime) => dispatch(TimeActionCreator.changeTime({ currentTime }));
  return <Home currentTime={currentTime} changeTime={changeTime } />;
}

export { Home };
export default HomeContainer;
```

7. 다중 리듀서

❖실행 결과

- 전체 상태 트리 확인





Q&A