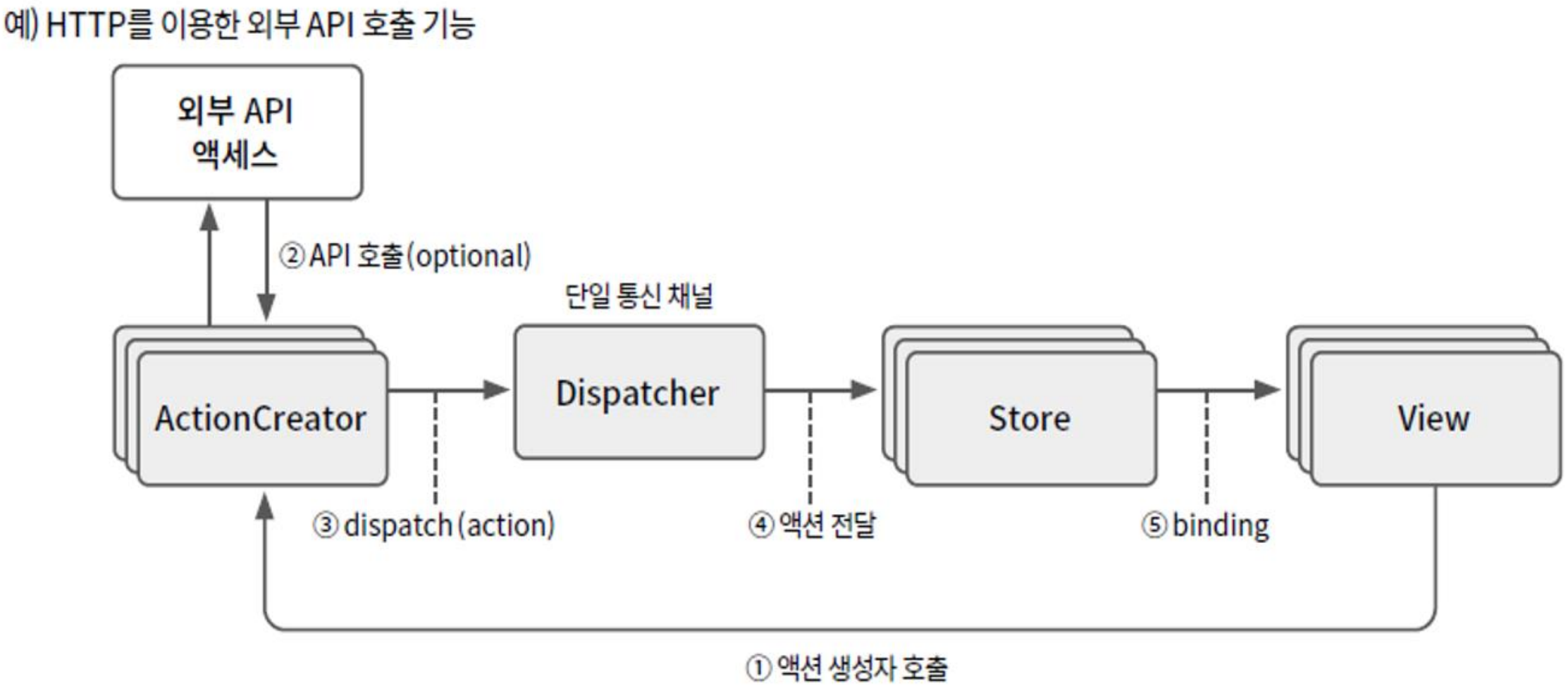
- redux + typescript

1. Redux 리뷰

❖Redux Architecture



1. Redux 리뷰

❖리덕스 구성요소

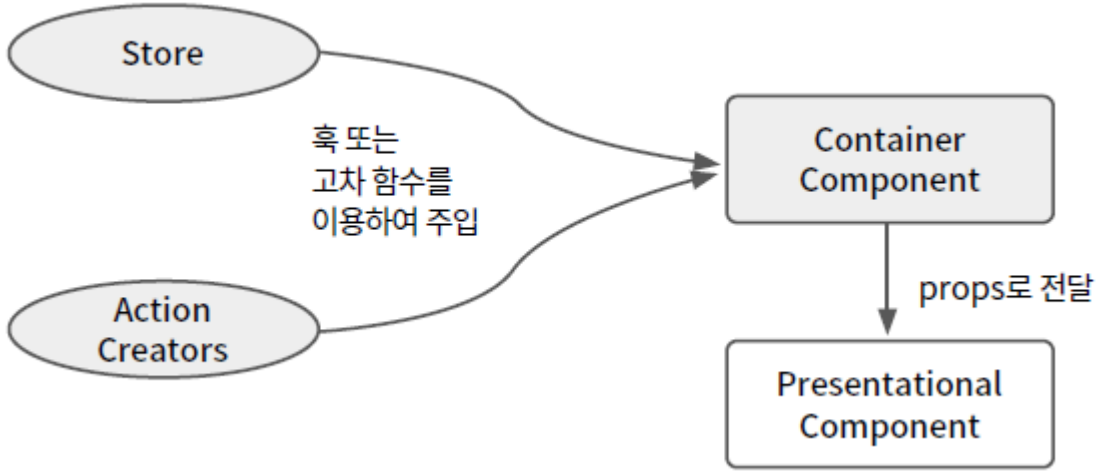
- 스토어
  - 단일 스토어 : 내부 상태는 읽기 전용(read only)
  - 모든 액션은 이 지점을 거쳐감
  - 이 지점만 관찰하면 상태 변경 이력, 데이터 흐름 등 상태 추적에 필요한 모든 중요한 정보를 획득할 수 있음
  - 애플리케이션 전체의 상태를 한 곳에서 관리하므로....
    - 상태(State)가 복잡해지고...
    - 상태를 변경하는 작업도 복잡해지고...
    - 따라서 상태만 스토어에서 관리! 상태 변경 작업은 리듀서에게 위임!
- 리듀서(Reducer)
  - 리듀서 : 기존 상태와 상태 변경에 필요한 값을 인자로 전달받아 새로운 상태를 만들어 리턴하는 함수
    - 불변성을 가지도록 새로운 상태를 생성해야 함
    - 상태 변경이외의 부작용(Side Effect: 예-외부 API 호출 등)가 존재해서는 안됨
  - 다중 리듀서 가능
    - 상태 변경 작업이 복잡할 때는 다중 리듀서로 구성
    - 자식 리듀서가 리턴한 값 상태 트리를 모아 부모가 전체 상태 트리를 조합하여 최종적으로 스토어로 리턴함

1. Redux 리뷰

- 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" } }

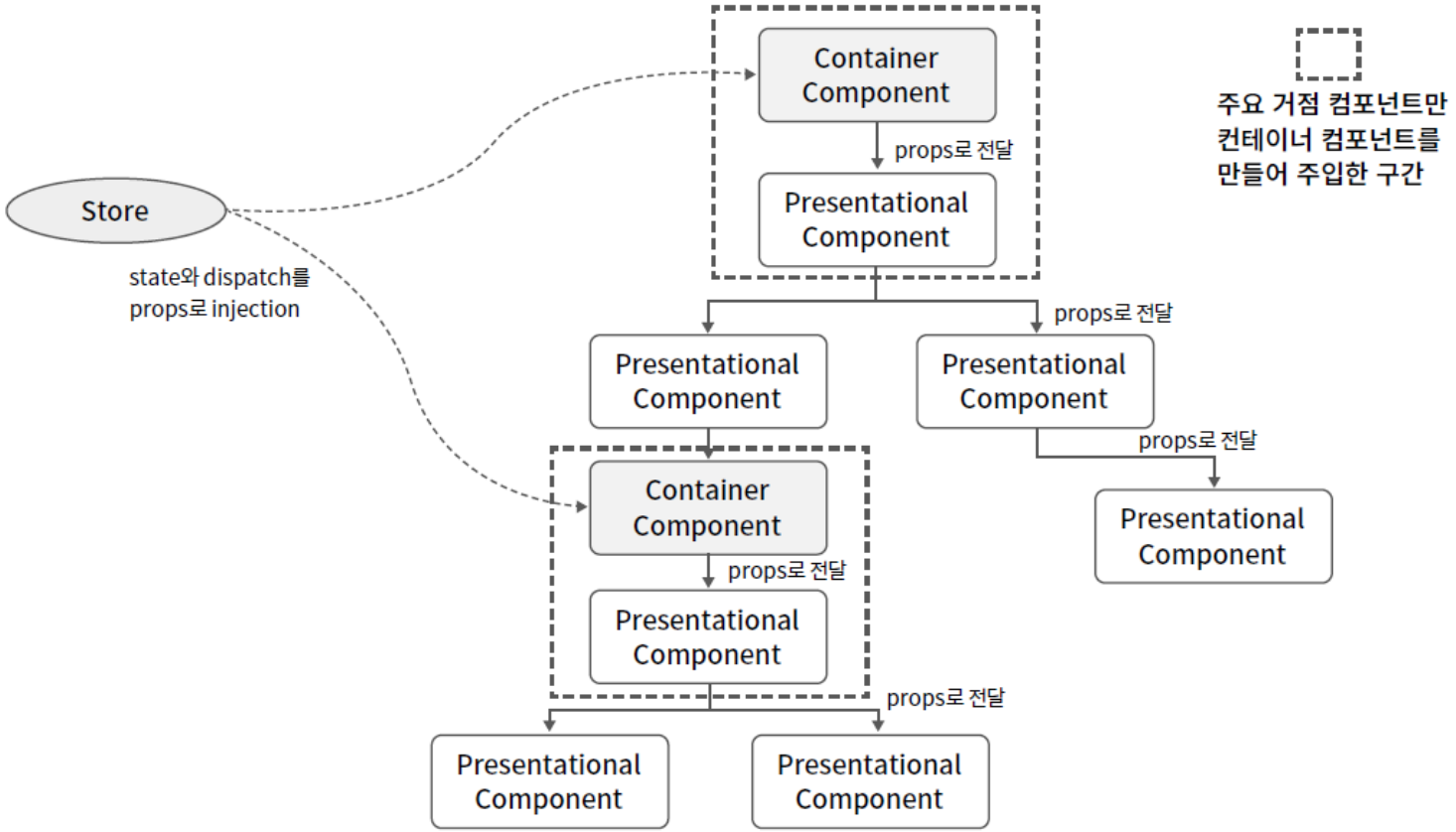
❖리덕스 컨테이너 컴포넌트

- 스토어와 연결되는 컴포넌트는 표현 컴포넌트로 작성할 수 있음
- react-redux 라이브러리
  - 표현컴포넌트 속성에 스토어의 상태를 주입(Injection)시켜주는 컨테이너 컴포넌트 손쉽게 만들수 있도록 함
    - connect() 고차함수
    - react-redux 훅 : useSelector() 등



1. Redux 리뷰

- 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



1. Redux 리뷰

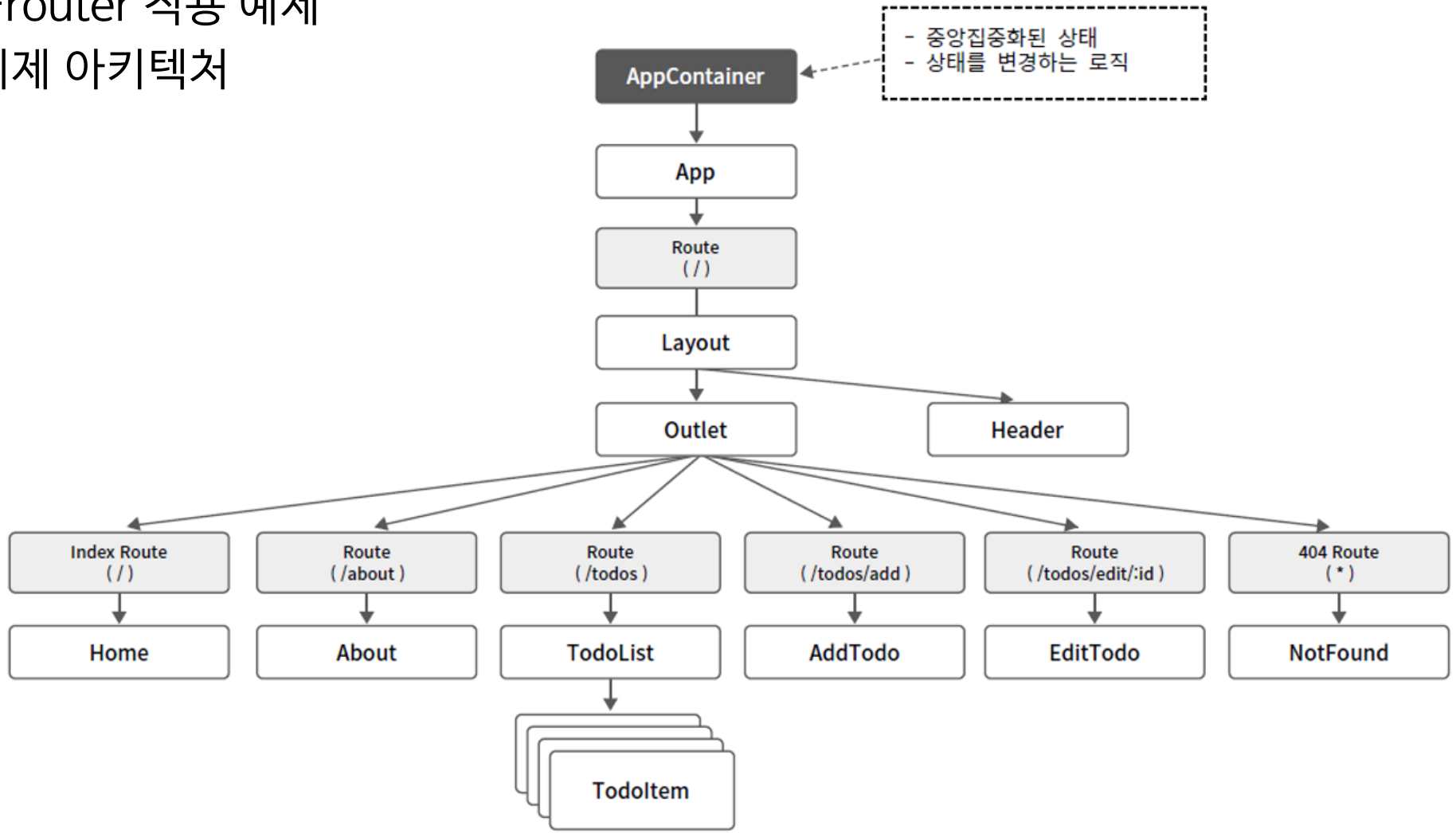
❖컨테이너 컴포넌트 생성

- react-redux가 제공하는 훅 사용
- useStore()
  - 리덕스 스토어 객체를 리턴. 스토어 객체의 모든 속성, 메서드를 이용하려면 이 훅을 사용함
- useDispatch()
  - 리덕스 스토어 객체의 dispatch() 함수만을 리턴함. 이 함수를 이용해 액션을 스토어로 전달할 수 있음
- useSelector()
  - 리덕스 스토어의 전체 상태 트리 중에서 일부 상태 정보를 리턴할 수 있음

2. Redux 기본 적용

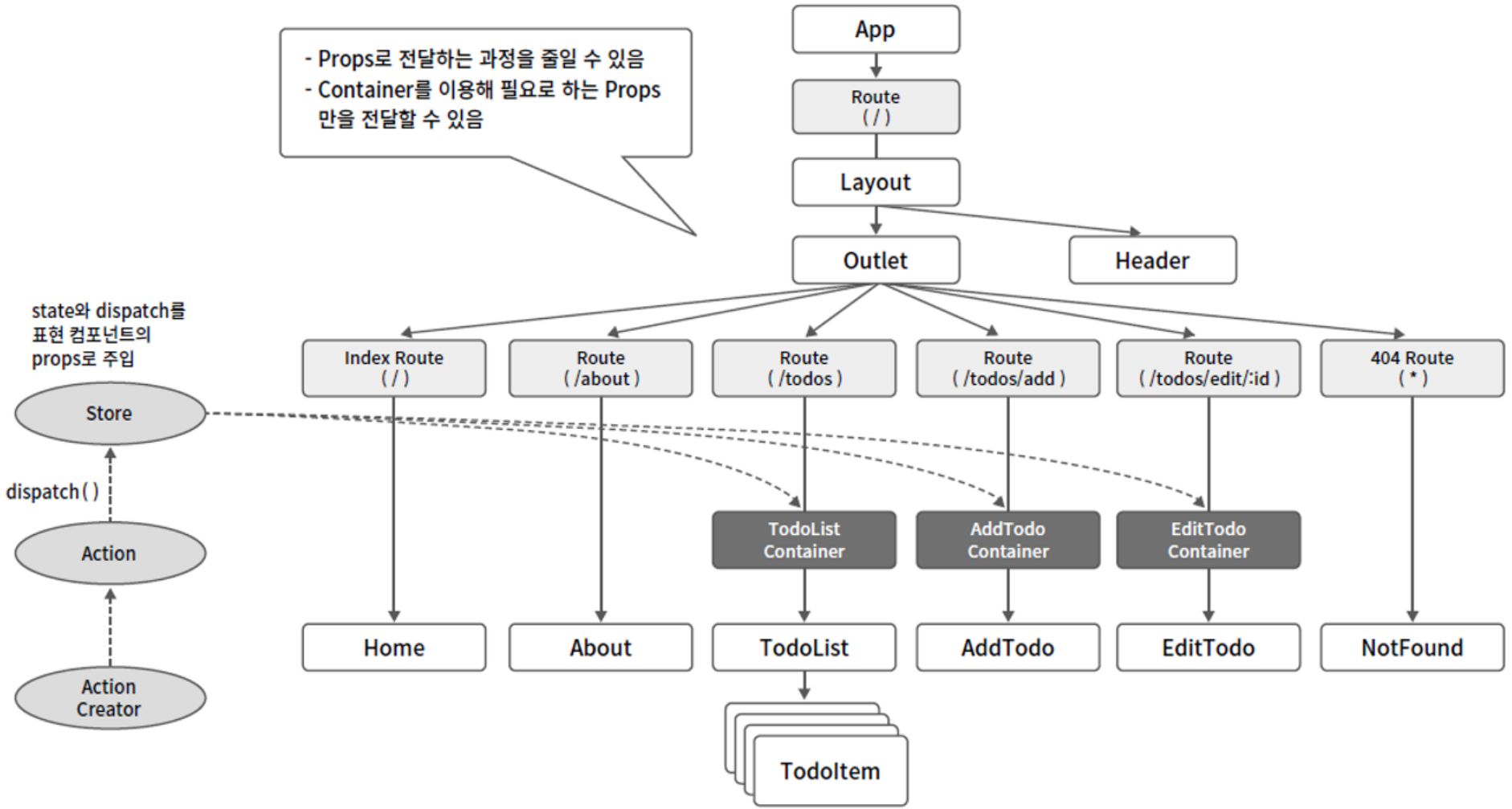
❖시작 예제 : todolist-app-router-0

- react-router 적용 예제
- 제공 예제 아키텍처



2. Redux 기본 적용

❖Redux 적용 후 예제 아키텍처



2. Redux 기본 적용

❖RTK 적용

❖패키지 설치

- npm install redux react-redux @reduxjs/toolkit

❖ Redux 구성요소 작성

- src/redux 폴더를 생성하고 폴더 내부에 다음 파일 생성
  - AppStore.ts, TodoActionCreator.ts, TodoReducer.ts
- src/redux/TodoActionCreator.ts 작성
  - createAction 메서드에 전달할 액션 페이로드 형식을 제네릭으로 정의해야 함

```
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;
```

2. Redux 기본 적용

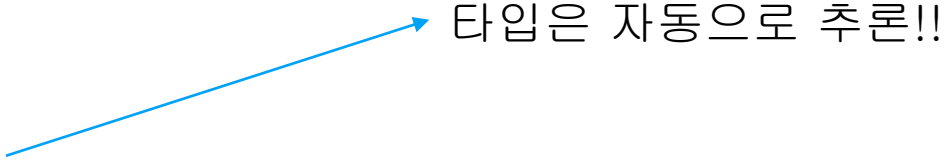
- src/redux/TodoReducer.ts 작성

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

//두 타입 모두 여러 컴포넌트, 모듈에서 재사용되므로 export!!
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 },
  ],
};

(이어서)
```

2. Redux 기본 적용

- src/redux/TodoReducer.ts 작성(이어서)

```
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 };
    });
});
export default TodoReducer;
```



2. Redux 기본 적용

- src/redux/AppStore.ts 작성

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

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

2. Redux 기본 적용

❖Container 컴포넌트

- AddTodo, EditTodo, TodoList 컴포넌트에 대해서만 Container 생성
- src/pages/AddTodo.tsx 변경

```
import { useState } from "react";
import { useDispatch } from "react-redux";
import { useNavigate } from "react-router";
import TodoActionCreator from "../redux/TodoActionCreator";
……(생략)

const AddTodo = ({ addTodo }: PropsType) => {
  ……(생략)
};

const AddTodoContainer = ()=>{
  const dispatch = useDispatch();
  const addTodo = (todo: string, desc: string) => dispatch(TodoActionCreator.addTodo({ todo, desc }));
  return <AddTodo addTodo={addTodo} />
}

export default AddTodoContainer;
export { AddTodo };
```

2. Redux 기본 적용

- src/pages/TodoList.tsx 변경

```
.....(생략)
import { useDispatch, useSelector } from "react-redux";
import { TodoItemType, TodoStatesType } from "../redux/TodoReducer";
import TodoActionCreator from "../redux/TodoActionCreator";

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

const TodoListContainer = () => {
  const dispatch = useDispatch();
  const todoList = useSelector((state:TodoStatesType)=>state.todoList);
  const toggleDone = (id: number) => dispatch(TodoActionCreator.toggleDone({ id }))
  const deleteTodo = (id: number) => dispatch(TodoActionCreator.deleteTodo({ id }))

  return <TodoList todoList={todoList} deleteTodo={deleteTodo} toggleDone={toggleDone} />
}

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

2. Redux 기본 적용

- src/redux/EditTodo.tsx 변겨

```
.....(생략)
import { useDispatch, useSelector } from "react-redux";
import { TodoItemType, TodoStatesType } from "../redux/TodoReducer";
import TodoActionCreator from "../redux/TodoActionCreator";

.....(생략)
const EditTodo = ({ todoList, updateTodo }: PropsType) => {
  .....(생략)
};

const EditTodoContainer = ()=>{
  const dispatch = useDispatch();
  const todoList = useSelector((state:TodoStatesType)=>state.todoList);
  const updateTodo = (id: number, todo: string, desc: string, done: boolean) => dispatch(
      TodoActionCreator.updateTodo({ id, todo, desc, done })
  )

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

export default EditTodoContainer;
export  { EditTodo };
```

2. Redux 기본 적용

- src/AppContainer.tsx 삭제
- src/App.tsx 변경
  - 속성 필요 없음, 자식 컴포넌트 속성 전달하지 않음

```
……
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;
```

2. Redux 기본 적용

- src/main.tsx 변경

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

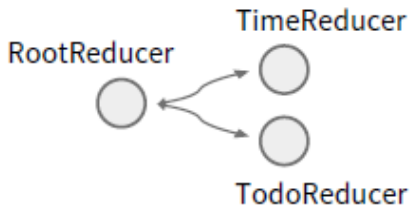
3. 다중 리듀서 적용

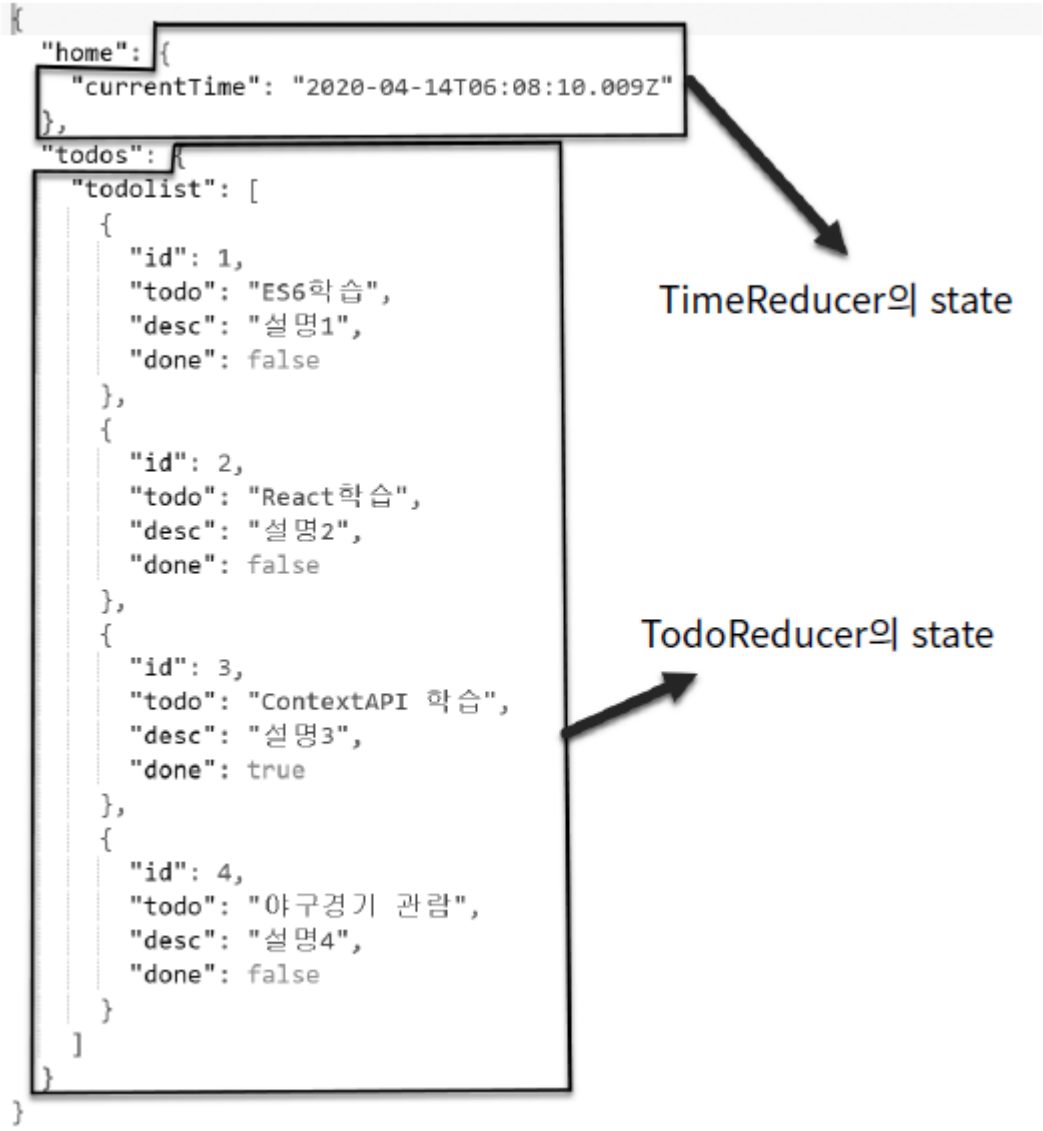
❖다중 리듀서 설명은 이미 진행한 바 있음
- Typescript 관련된 부분에 집중

❖애플리케이션의 상태가 복잡해지면?
- 리듀서의 상태 변경 기능도 많아지고 복잡해짐
  - 하나의 리듀서로 처리 불가능
  - 따라서 여러 개의 리듀서로 분리시켜야 함

❖다중 리듀서를 사용하려면...
- 상태 트리를 꼼꼼하게 설계해야 함
- 자식 리듀서들은 전체 상태 트리 중 특정 하위의 트리를 담당하기 때문에...

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





3. 다중 리듀서 적용

❖다중 리듀서 테스트

- 기능을 확인하기 위해 Todolist 예제에 새로운 컴포넌트 추가와 약간의 코드 추가
  - TimeReducer
  - RootReducer : TimeReducer와 TodoReducer를 결합한 Root Reducer
  - TimeActionCreator
  - AppStore : 두 자식 리듀서를 결합하도록 변경
  - MyTime 컴포넌트
  - Home 컴포넌트 변경
- 추가할 상태와 Dispatch 메서드
  - currentTime, changeTime()

❖Typescript 적용시 신경써야 할 부분

- 상태트리가 변경되어야 하므로 상태에 대한 Type 지정
- 자식 리듀서가 사용하는 상태 Type을 조합하여 루트 상태의 Type을 선언함

3. 다중 리듀서 적용

❖src/redux/TimeActionCreator.ts 추가

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

❖src/redux/TimeReducer.ts 추가

```
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;
```

3. 다중 리듀서 적용

❖src/redux/AppStore.ts 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,
});

//currentTime 이 Date 타입이므로 이 타입의 값을 직렬화할 때의 경고를 막기 위해 미들웨어 속성 추가
const AppStore = configureStore({
  reducer: RootReducer,
  middleware: (getDefaultMiddleware) => {
    return getDefaultMiddleware({ serializableCheck: false })
  },
});

export default AppStore;
```

3. 다중 리듀서 적용

❖src/pages/EditTodo.tsx 변경

- 전체 상태 트리가 변경되었으므로 표현 컴포넌트로 주입시킬 상태 속성을 변경하도록 수정

```
……(생략)
import { RootStatesType } from "../redux/AppStore";
……(생략)

const EditTodo = ({ todoList, updateTodo }: PropsType) => {
   ……(생략)
};

const EditTodoContainer = ()=>{
  const dispatch = useDispatch();
  const todoList = useSelector((state:RootStatesType)=>state.todos.todoList);
  const updateTodo =
       (id: number, todo: string, desc: string, done: boolean) => dispatch(TodoActionCreator.updateTodo({ id, todo, desc, done }))

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

export default EditTodoContainer;
export  { EditTodo };
```

3. 다중 리듀서 적용

❖src/pages/TodoList.tsx 변경

```
……(생략)
import { RootStatesType } from "../redux/AppStore";
……(생략)

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

const TodoListContainer = () => {
  const dispatch = useDispatch();
  const todoList = useSelector((state:RootStatesType)=>state.todos.todoList);
  const toggleDone = (id: number) => dispatch(TodoActionCreator.toggleDone({ id }))
  const deleteTodo = (id: number) => dispatch(TodoActionCreator.deleteTodo({ id }))

  return <TodoList todoList={todoList} deleteTodo={deleteTodo} toggleDone={toggleDone} />
}

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

3. 다중 리듀서 적용

❖src/pages/MyTime.tsx 추가

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

const MyTime = ({ currentTime, changeTime }: PropsType) => {
  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;
```

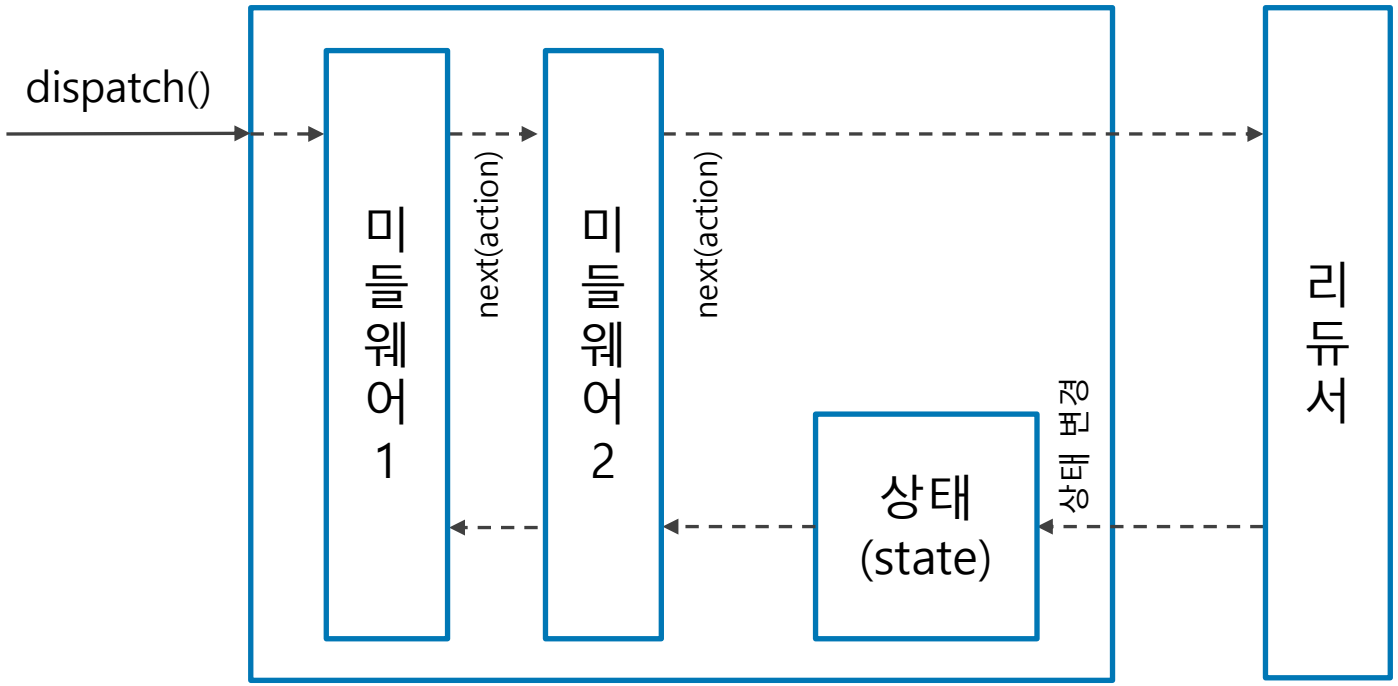
3. 다중 리듀서 적용

❖src/pages/Home.tsx 변경

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

type PropsType = {  currentTime: Date;  changeTime: (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 = (currentTime: Date)=>dispatch(TimeActionCreator.changeTime({currentTime}));
  return <Home currentTime={currentTime} changeTime={changeTime} />
}
export default HomeContainer;
export { Home };
```

4. 미들웨어

❖미들웨어 리뷰




```
//미들웨어 함수 구조
(store) => (next) => (action) => {
   ..... (전)
   next(action)
   ..... (후)
}
```

4. 미들웨어

❖Middleware Type

```
import { Middleware, combineReducers } from "redux";
.....
//store, next, action 인자는 모두 타입 추론으로...
const logger: Middleware = (store) => (next) => (action) => {
   console.log('전달된 액션 : ', action);
   console.log('리듀서 실행 전 상태 : ', store.getState());
   next(action);
   console.log('리듀서 실행 후 상태 : ', store.getState());
};

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

export default AppStore;
```

5. UtilityType,타입이동기법 이용

❖직전 예제까지의 문제점

- 데이터 생성하는 곳에서 매번 타입을 선언하는 것
  - TimeReducer, TodoReducer, AppStore 등 여러 곳에서 타입이 선언되었음
- 데이터가 생성되어 모이는 곳에서 데이터의 타입을 추론하여 선언하는 것은 어떨까?
  - Store 한 곳에서 사용하는 타입을 관리하는 것

❖이를 위해 Typescript를 학습할 때 배웠던 내용을 활용해 보자

- Utility Type
  - ReturnType<Type>
  - Pick<Type, Keys>
  - Omit<Type, Keys>
- 타입 이동 기법
  - typeof 키워드를 이용한 이동
  - 특정 타입의 하위 멤버 타입 이동 : 예) Member["name"]

5. UtilityType,타입이동기법 이용

❖코드 리팩토링

- src/redux/TimeReducer.ts 변경 : 타입 선언 부분을 모두 삭제함

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

//type을 선언하고 export하는 부분을 제거함

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

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

export default TimeReducer;
```

5. UtilityType,타입이동기법 이용

- src/redux/TodoReducer.ts 변경 : 타입 선언 부분을 모두 삭제함

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

//type을 선언하고 export하는 부분을 제거함

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 },
  ],
};

const TodoReducer = createReducer(initialState, (builder) => {
   ……(생략)
});

export default TodoReducer;
```

5. UtilityType,타입이동기법 이용

- src/redux/AppStore.ts 변경
  - 필요한 타입을 이곳에서 생성하여 export 함

```
……(생략)

//TimeStatesType, TodoStates 타입을 임포트하여 조합하여 RootStatesType을 선언하는 부분 삭제

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


……
const AppStore = configureStore({
   ……(생략)
});

//스토어의 dispatch 타입 이동
export type AppDispatch = typeof AppStore.dispatch
//스토어의 getState() 함수의 리턴값으로 타입 지정
export type RootStatesType = ReturnType<typeof AppStore.getState>;
//RootStatesType의 todos.todoList 하위 속성의 타입 이동
export type TodoItemType = RootStatesType["todos"]["todoList"][0];

export default AppStore;
```

5. UtilityType,타입이동기법 이용

❖각 컴포넌트에서 참조하는 TodoItemType, RootStatesType을 변경함

- 변경 대상 : TodoList, EditTodo, AddTodo
- 예시) TodoList 컴포넌트

```
……(생략)
import { AppDispatch, RootStatesType, TodoItemType } from "../redux/AppStore";
……(생략)

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

const TodoListContainer = () => {
  const dispatch = useDispatch<AppDispatch>();
  const todoList = useSelector((state:RootStatesType)=>state.todos.todoList);
  const toggleDone = (id: number) => dispatch(TodoActionCreator.toggleDone({ id }))
  const deleteTodo = (id: number) => dispatch(TodoActionCreator.deleteTodo({ id }))

  return <TodoList todoList={todoList} deleteTodo={deleteTodo} toggleDone={toggleDone} />
}

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

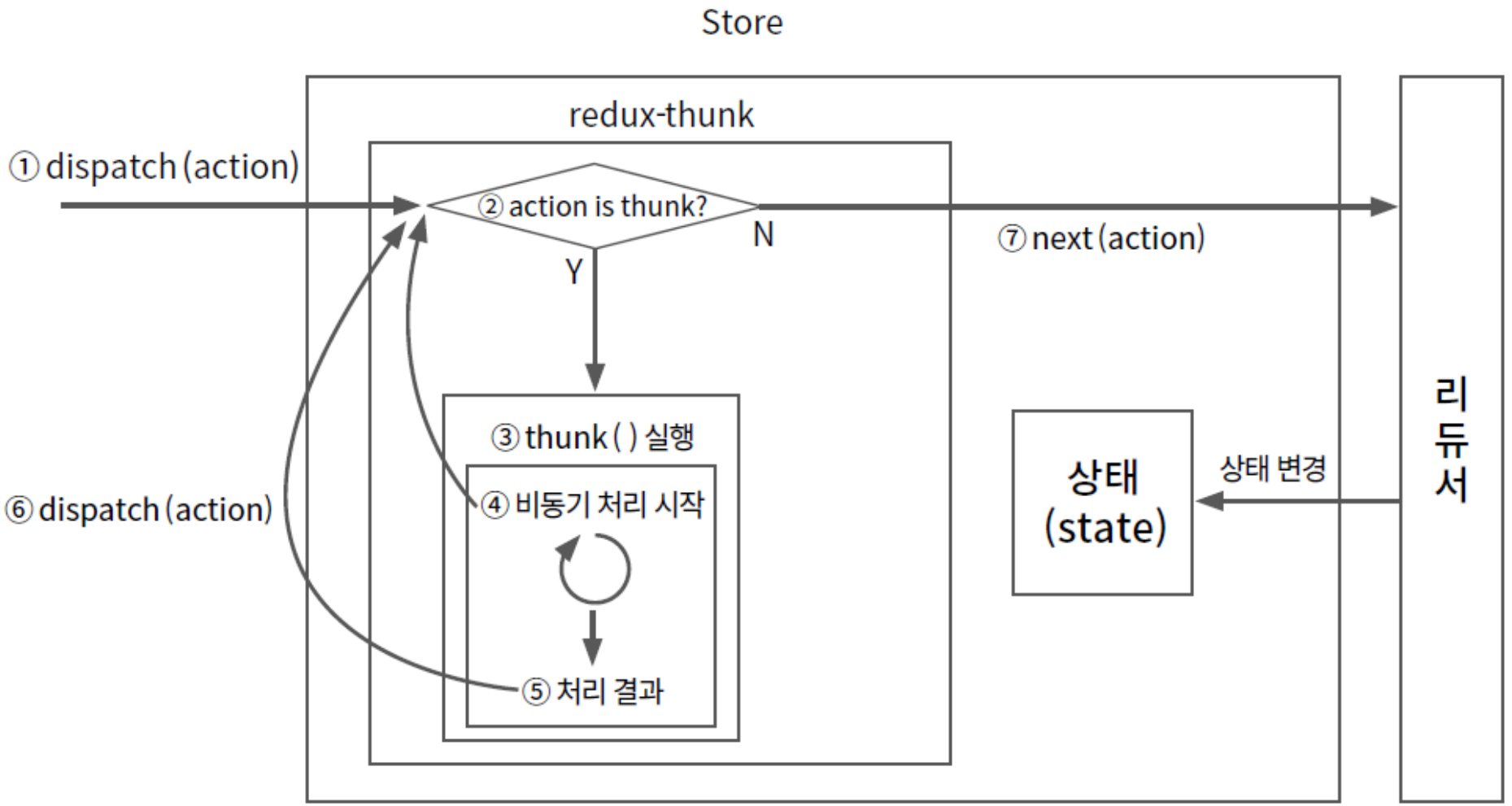
5. UtilityType,타입이동기법 이용

❖주의사항

- 추론된 타입으로 타입 이동 기법을 항상 적용할 수 있는 것은 아님
- 초기 상태가 정확하게 주어진 것이 있다면 초기 상태를 이용해서 추론하고 타입 이동
- 초기 상태가 없다면 타입 추론을 할 수 없으므로 타입 이동이 불가능

6.1 redux-thunk + RTK

❖redux-thunk 미들웨어 아키텍처



6.1 redux-thunk + RTK

❖일반적인 thunk 함수 형태

```
//thunk 함수의 일반적인 패턴(async/await 버전)
async (dispatch: ThunkDispatch<{}, {}, AnyAction>) => {
  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());
  }
};
```

6.1 redux-thunk + RTK

❖RTK
- redux-thunk 패키지 이미 포함
- redux-thunk를 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

6.1 redux-thunk + RTK

❖createAsyncThunk 사용 형태

```
const asyncAction = createAsyncThunk("액션명", async (arg, thunkAPI) => {
  //args는 비동기 처리할 때 필요한 아규먼트입니다.
  //비동기 처리 후 마지막에 리턴하는 값이
  //최종적으로 완료했을 때 전달하는 action의 페이로드가 됩니다.
return payload
})
```

- 두번째 인자 함수 : payloadCreator라는 비동기 처리 수행 함수
  - 요청/응답 시점별로 dispatch(action)하지 않아도 됨
  - 만일 직접 dispatch하고 싶다면 thunkAPI 인자를 이용하여 dispatch, fulfillWithValue, rejectWithValue, getState 등의 함수를 이용해 상태를 확인하고 액션을 직접 전달할 수 있음

Note

payloadCreator 함수는 Promise 기반!

createAsyncThunk( ) 함수에 전달되는 두 번째 인자인 payloadCreator 함수는 Promise 기반입니다. 따라서 async/await이나 Promise 기반으로 작성해야 합니다.

6.2 redux-thunk + RTK + Typescript

❖createAsyncThunk 함수의 제네릭 타입

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

- 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
  - payloadCreator 함수의 두번째 인자인 thunkAPI 인자의 각 속성에 지정할 타입
  - dispatch?, state?, rejectValue?, extra?

6.2 redux-thunk + RTK + Typescript

❖createAsyncThunk 함수의 제네릭 타입

- 코드 예시

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

const actionSearchContacts = createAsyncThunk<
  { contacts: ContactItemType[] },
  { name: string },
  {
    dispatch: AppDispatch,
    state: ContactStateType,
    rejectValue: ThunkErrorType,
  }
>(
  "searchContacts",
  async ({ name }, thunkAPI) => {
    try {
      let 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 })
    }
  }
);
```

6.2 redux-thunk + RTK + Typescript

❖예제 프로젝트 생성

- npm init vite contacts-clients-ts -- --template react-ts
- cd contacts-client-ts
- 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; }
```

6.2 redux-thunk + RTK + Typescript

❖src/redux/ContactAction.ts 작성

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

// 향후 스토어에 타입 이동으로 변경할 수 있음.
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
}
```

6.2 redux-thunk + RTK + Typescript

❖src/redux/ContactAction.ts 작성

```
export const actionSearchContacts = createAsyncThunk<
  { contacts: ContactItemType[] },
  { name: string },
  {
    dispatch: AppDispatch,
    state: ContactStateType,
    rejectValue: ThunkErrorType,
  }
>(
  "searchContacts",
  async ({ name }, thunkAPI) => {
    try {
      let 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 })
    }
  }
);
```

6.2 redux-thunk + RTK + Typescript

❖src/redux/ContactReducer.ts 작성

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

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

const ContactReducer = createReducer(initialState, (builder) => {
  builder
    .addCase(actionSearchContacts.pending, (state, action) => {
      state.isLoading = true;
      state.status = action.meta.arg.name + " 포함 이름으로 조회중";
    })
    .addCase(actionSearchContacts.fulfilled, (state, action) => {
      state.contacts = action.payload.contacts;
      state.isLoading = false;
      state.status = "조회 완료";
    })
    .addCase(actionSearchContacts.rejected, (state) => {
      state.contacts = [];
      state.isLoading = false;
      state.status = "조회 실패";
    });
});

export default ContactReducer;
```

6.2 redux-thunk + RTK + Typescript

❖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);
}

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

export default ContactStore;
```

6.2 redux-thunk + RTK + Typescript

❖src/App.tsx 작성

```
import { useState } from "react";
import { useDispatch, useSelector } from "react-redux";
import { AppDispatch, ContactItemType, ContactStateType, actionSearchContacts } 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>();
  var propsObject = {
    isLoading: useSelector((state: ContactStateType) => state.isLoading),
    status: useSelector((state: ContactStateType) => state.status),
    contacts: useSelector((state: ContactStateType) => state.contacts),
    searchContacts: (name: string) => dispatch(actionSearchContacts({ name }))
  };
  return <App {...propsObject} />;
};

export default AppContainer;
```

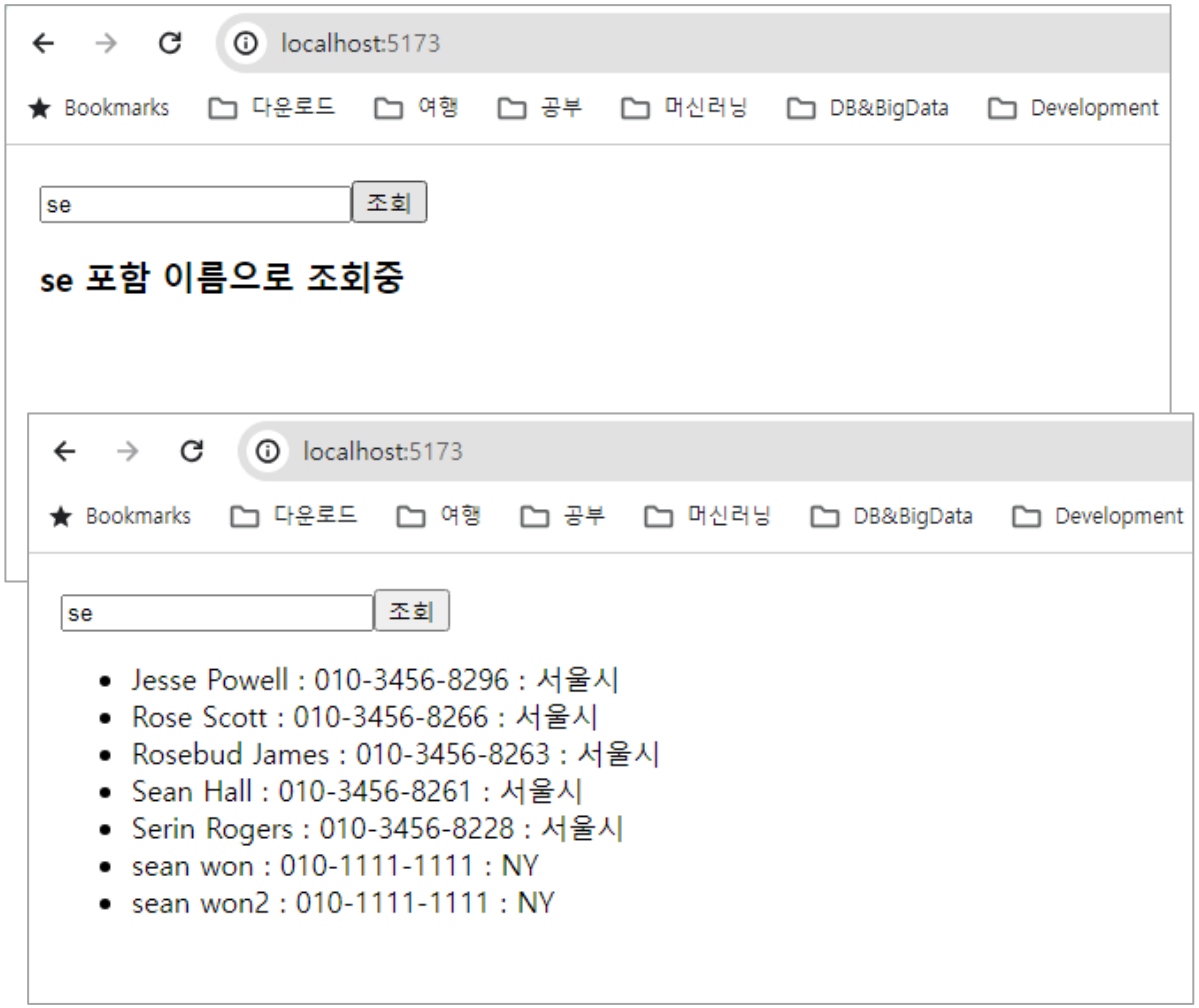
6.2 redux-thunk + RTK + Typescript

❖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>,
)
```

6.2 redux-thunk + RTK + Typescript

❖실행 결과




```
action
▼ {type: 'searchContacts/pending', payload: undefined, meta: {…}}
  ▼ meta:
    ▶ arg: {name: 'se'}
      requestId: "nb498nhIHB_HCg-m-G33y"
      requestStatus: "pending"
    ▶ [[Prototype]]: Object
    payload: undefined
    type: "searchContacts/pending"
  ▶ [[Prototype]]: Object
action  ▼ {type: 'searchContacts/fulfilled', payload: {…}, meta: {…}}
        ▼ meta:
          ▶ arg: {name: 'se'}
            requestId: "nb498nhIHB_HCg-m-G33y"
            requestStatus: "fulfilled"
          ▶ [[Prototype]]: Object
        ▶ payload: {contacts: Array(7)}
          type: "searchContacts/fulfilled"
        ▶ [[Prototype]]: Object
```

Q&A