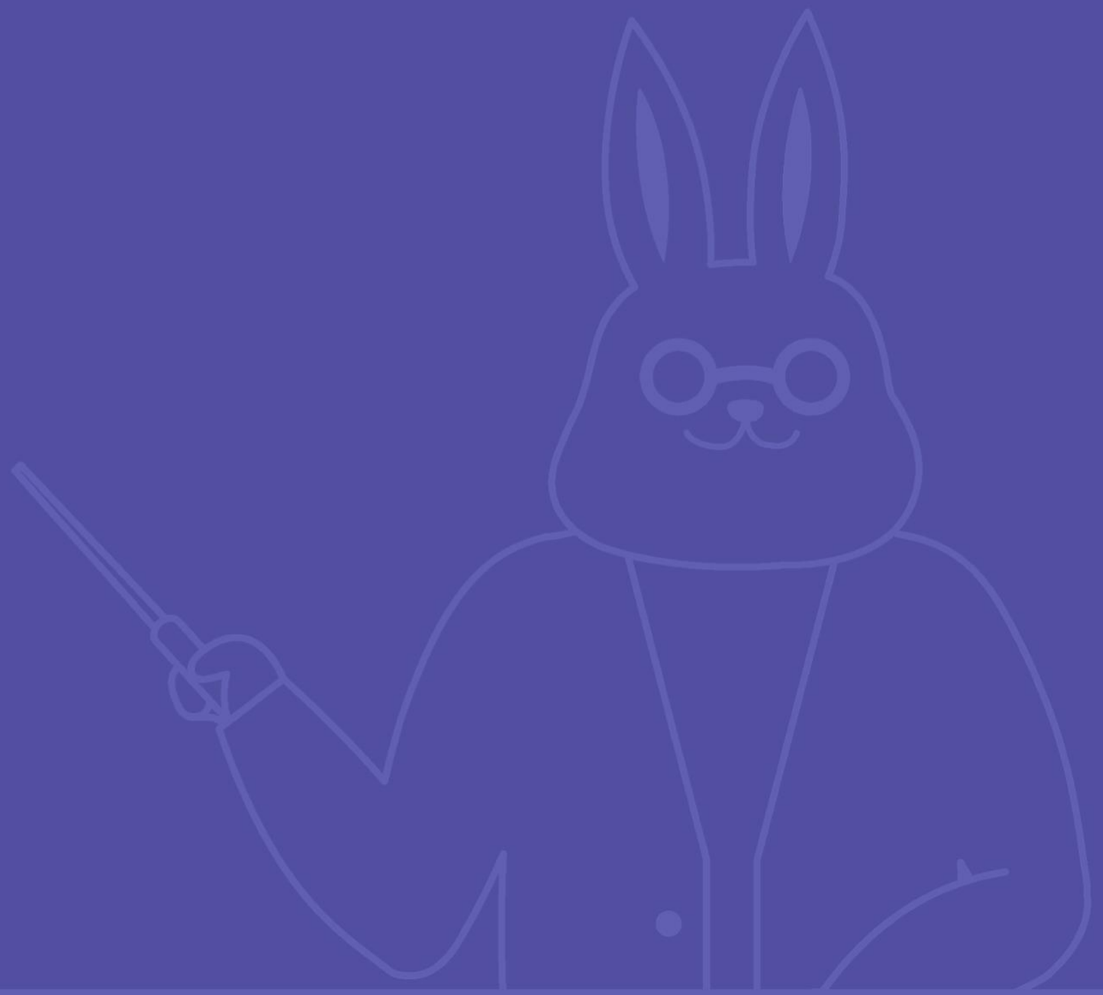
Adv. 1

React 심화 I

04 상태 관리

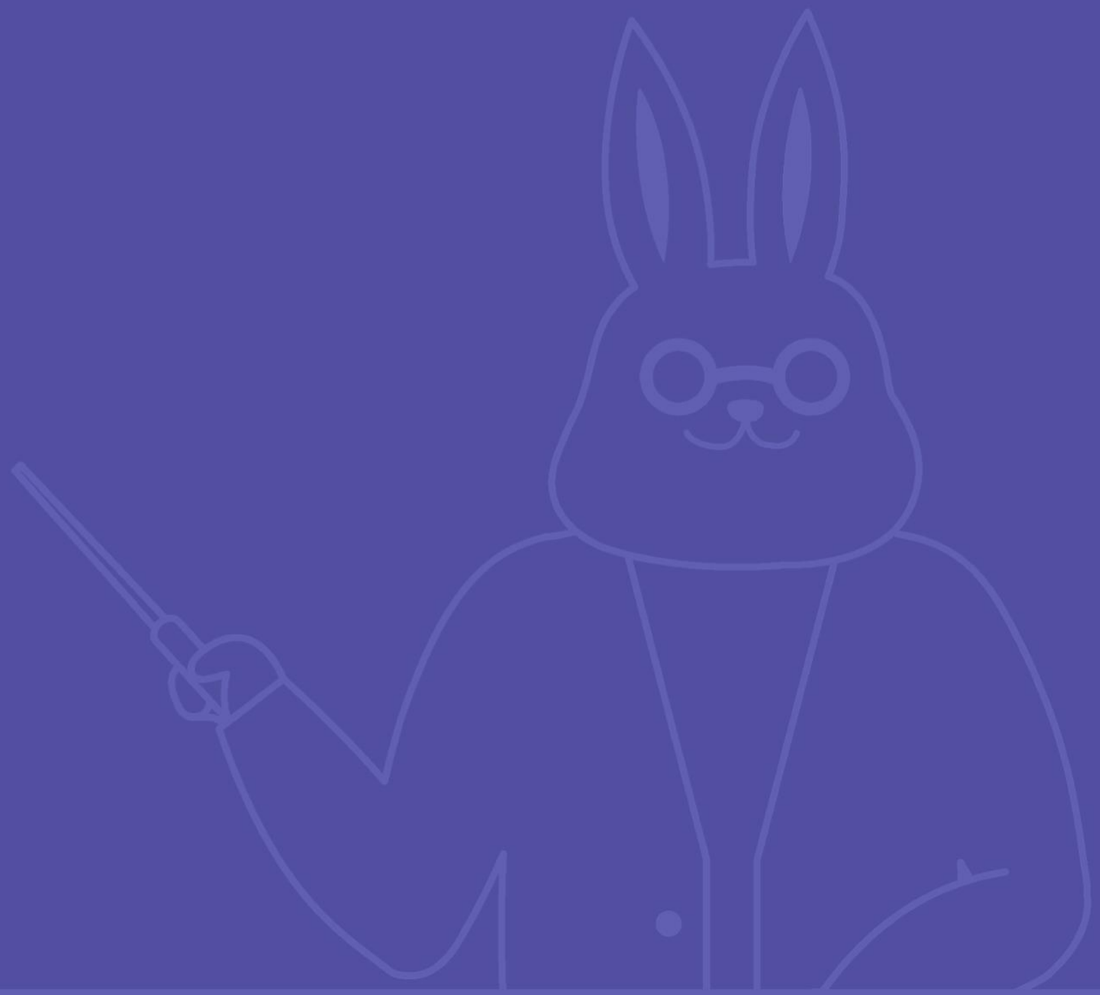


/* elice */

목차

01

상태 관리



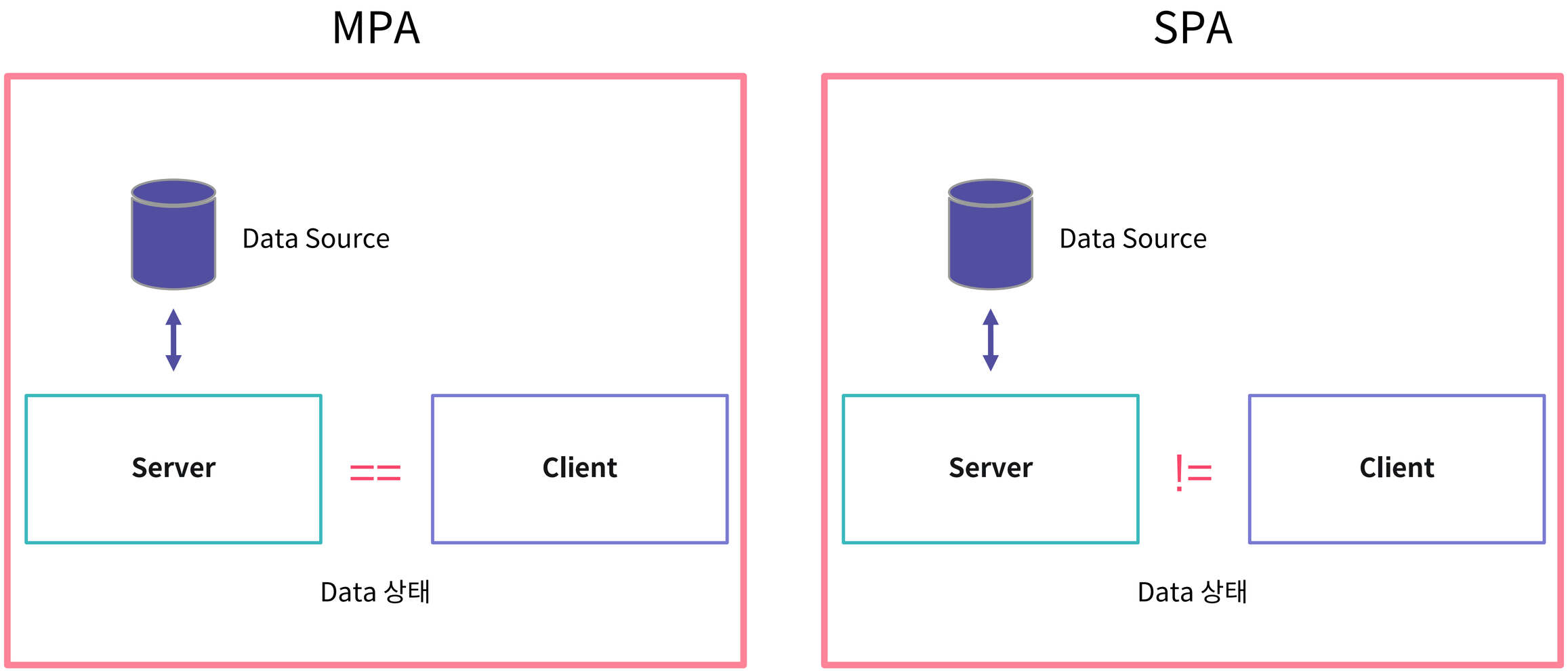
상태 관리

- 상태 관리 기술이란 앱 상에서의 데이터를 메모리 등에 저장하고 하나 이상의 컴포넌트에서 데이터를 공유하는 것.
- 한 컴포넌트 안에서의 상태, 여러 컴포넌트 간의 상태, 전체 앱의 상태 관리를 모두 포함.

MPA와 SPA에서의 상태 관리

- MPA에서는 서버의 데이터를 이용해 페이지를 렌더링하므로, 클라이언트의 데이터와 서버의 데이터가 큰 차이를 가지지 않음.
- SPA에서는 자체적으로 데이터를 갖고, 서버와의 동기화가 필요한 데이터만을 처리. 그 외의 데이터는 Client만의 데이터로 유지.

MPA와 SPA에서의 상태 관리



상태 관리 기술의 도입

- 상태가 많지 않거나, 유저와의 인터렉션이 많지 않다면
  매 작업 시 서버와 동기화하더라도 충분함.
- 앱이 사용하는 데이터가 점점 많아지고, 유저와의 인터렉션 시 임시로 저장하는
  데이터가 많아지는 경우 상태 관리를 고려.
- 프론트엔드 뿐만 아니라, 백엔드와의 데이터 통신도 충분히 고려해야 함.
  ex) GraphQL

상태 관리 기술의 장점

- 높은 품질의 코드를 작성하는 데 유리.
- 성능 최적화, 네트워크 최적화 등에 유리.
- 데이터 관리의 고도화.

  ex) localStorage 활용한 persist state

상태 관리 기술의 단점

- Boilerplate 문제.
- 파악해야 할 로직과 레이어가 많아짐.
- 잘못 사용할 경우, 앱의 복잡도만을 높이거나, 성능을 악화.
  ex) global state의 잘못된 활용 시 앱 전체 리렌더링 발생.

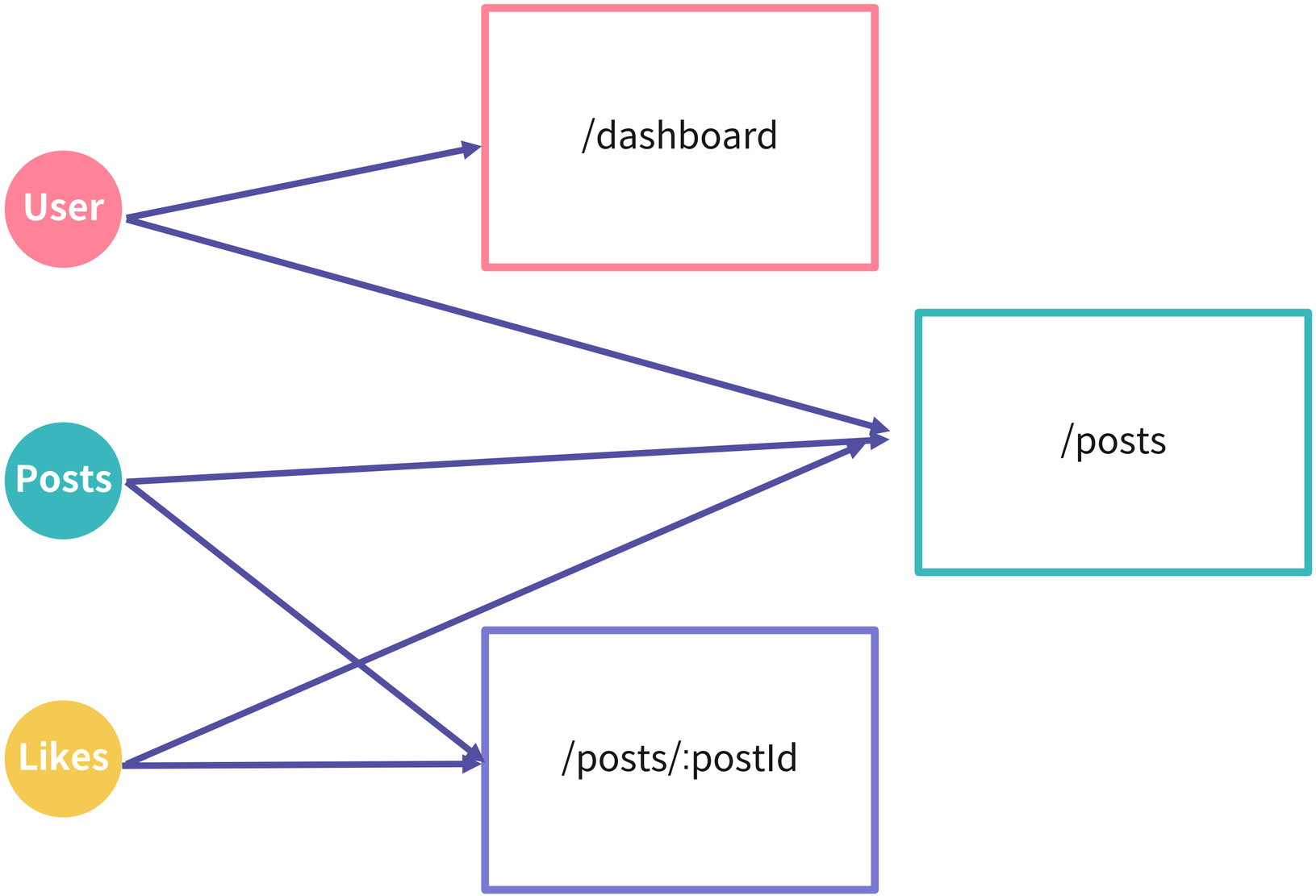
02

상태 관리 기술이 해결하는 문제들

데이터 캐싱과 재활용

- SPA에서 페이지 로딩 시마다 모든 데이터를 로딩한다면, 사용자 경험 측면에서 MPA를 크게 넘어서기 힘듦.
- 오히려 네트워크 요청 수가 많아져 더 느릴 수도 있음.
- 변경이 잦은 데이터가 아니라면, 데이터를 캐싱하고 재활용함.
- 변경이 잦다면, 데이터의 변경 시점을 파악해 최적화.
  ex) 일정 시간마다 서버에 저장, 타이핑 5초 후 서버에 저장

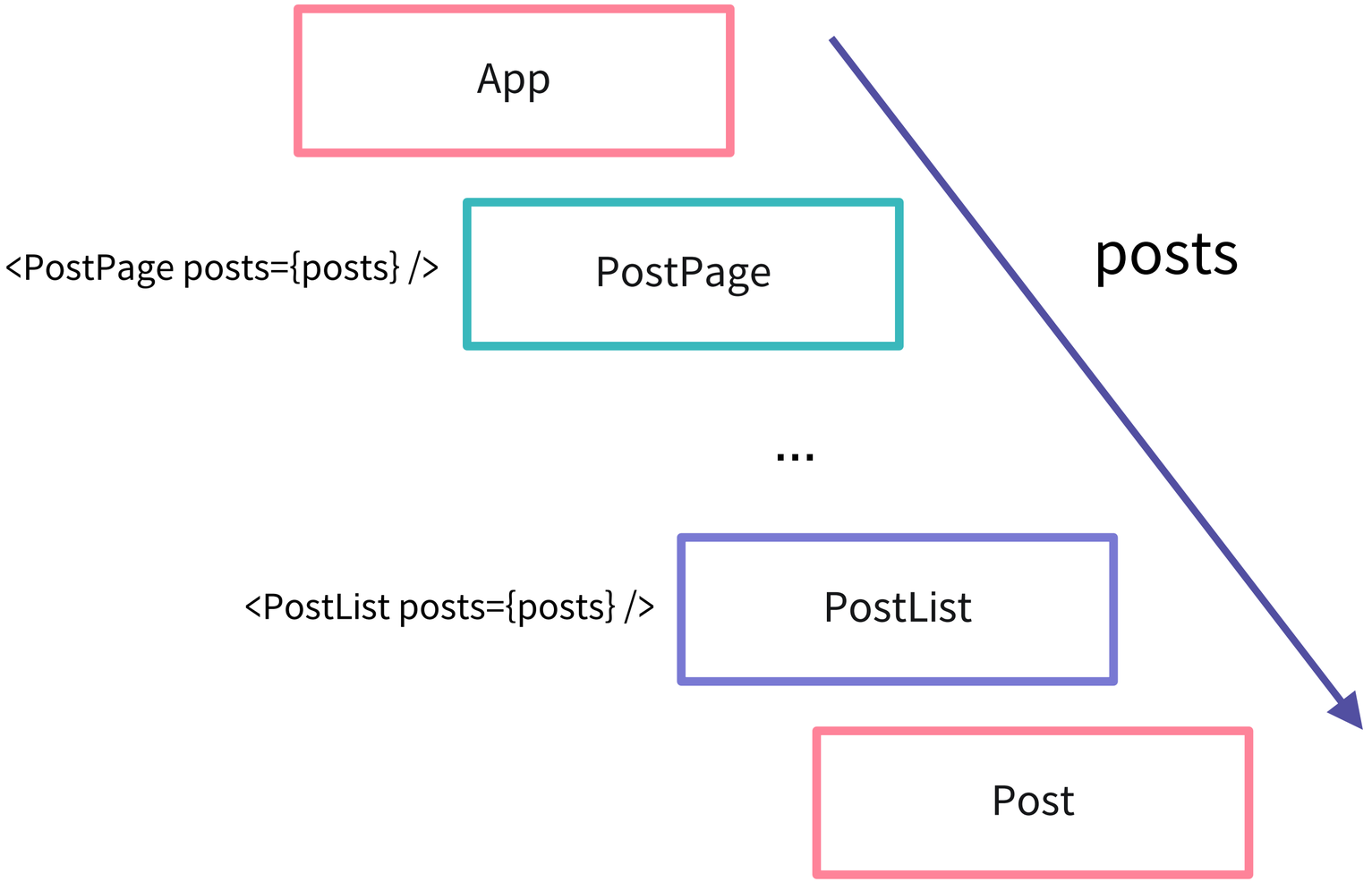
데이터 캐싱과 재활용



Prop Drilling

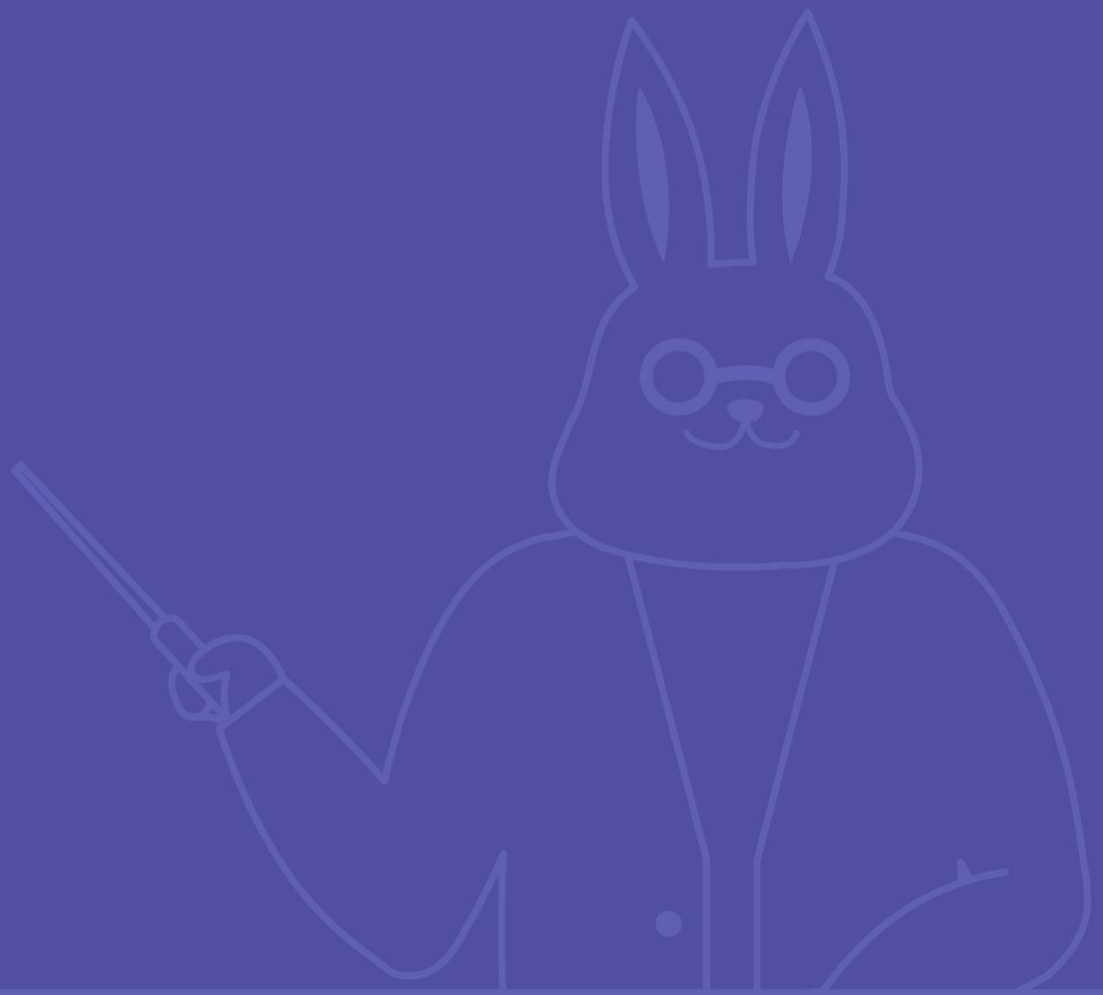
- 컴포넌트가 복잡해지는 경우, 상위 부모와 자식 컴포넌트 간의 깊이가 커짐.
- 최하단의 자식 컴포넌트가 데이터를 쓰기 위해
  최상위 컴포넌트부터 데이터를 보내야 하는 상황이 발생.
- Context API 등을 활용, 필요한 컴포넌트에서 데이터를 가져올 수 있음.
- 컴포넌트 간의 결합성을 낮춤.

Prop Drilling



03

Flux Pattern



Flux Pattern

- 2014년에 Facebook에서 제안한 웹 애플리케이션 아키텍처 패턴.
- Unidirectional data flow를 활용, 데이터의 업데이트와 UI 반영을 단순화.
- React의 UI 패턴인 합성 컴포넌트와 어울리도록 설계.
- redux, react-redux 라이브러리의 Prior art.

Flux Pattern vs MVC Pattern

- MVC 패턴에서는, View에서 특정 데이터를 업데이트하면 연쇄적인 업데이트가 일어남.
- 앱이 커지면 업데이트의 흐름을 따라가기 힘듦.
- Flux는 하나의 Action이 하나의 Update만을 만들도록 함.

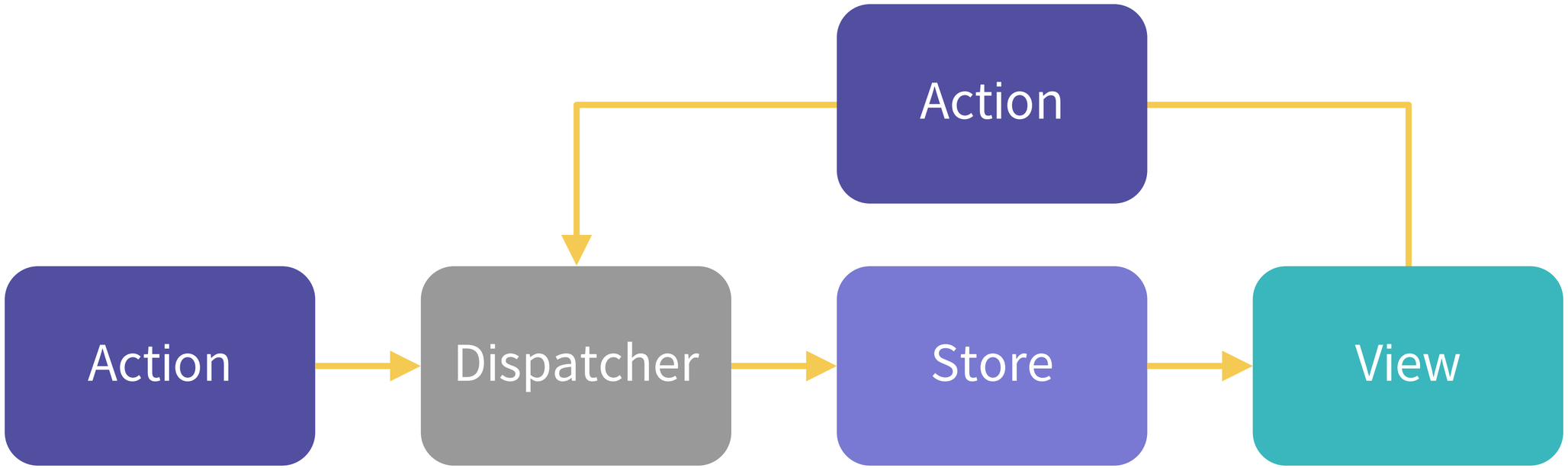
Flux Pattern vs MVC Pattern

- MVC 패턴에서는, View에서 특정 데이터를 업데이트하면 연쇄적인 업데이트가 일어남.
- 특정 유저의 인터렉션이 여러 UI 컴포넌트가 사용하는 데이터에 영향을 줄 때,
  MVC만으로는 앱의 복잡도를 낮추거나 업데이트의 흐름을 따라가기 어려움.
- Flux는 하나의 유저 인터렉션 당 하나의 Update만을 만들도록 함.
- data와 업데이트가 한 방향으로 흐르므로 UI의 업데이트를 예측하기 쉬움.

Flux 구조

- Action -> Dispatcher -> Store -> View 순으로 데이터가 흐름.
- store는 미리 dispatcher에 callback을 등록해, 자신이 처리할 action을 정의.
- action creator는 action을 생성하여 dispatcher로 보냄.
- dispatcher는 action을 store로 넘김.
- store는 action에 따라 데이터를 업데이트 후, 관련 view로 변경 이벤트 발생.
- View는 그에 따라 데이터를 다시 받아와 새로운 UI를 만듦.
- 유저 인터렉션이 발생하면 View는 action을 발생.

Flux 구조



04

useState, useRef, useContext, useReducer

상태 관리에 사용되는 훅

- 외부 라이브러리 없이 React가 제공하는 훅 만으로 상태 관리를 구현하기 위해 사용.
- 함수형 컴포넌트에 상태를 두고, 여러 컴포넌트 간 데이터와 데이터 변경 함수를 공유하는 방식으로 상태를 관리하게 됨.

useState

- 단순한 하나의 상태를 관리하기에 적합함.
- const [state, setState] = useState(initState | initFn)
- state가 바뀌면, state를 사용하는 컴포넌트를 리렌더함.
- useEffect와 함께, state에 반응하는 훅을 구축.

useRef

- 상태가 바뀌어도 리렌더링하지 않는 상태를 정의함.
- 즉, 상태가 UI의 변경과 관계없을 때 사용.
  ex) setTimeout의 timerId 저장
- uncontrolled component의 상태를 조작하는 등, 리렌더링을 최소화하는 상태 관리에 사용됨.
  ex) Dynamic Form 예시

useContext

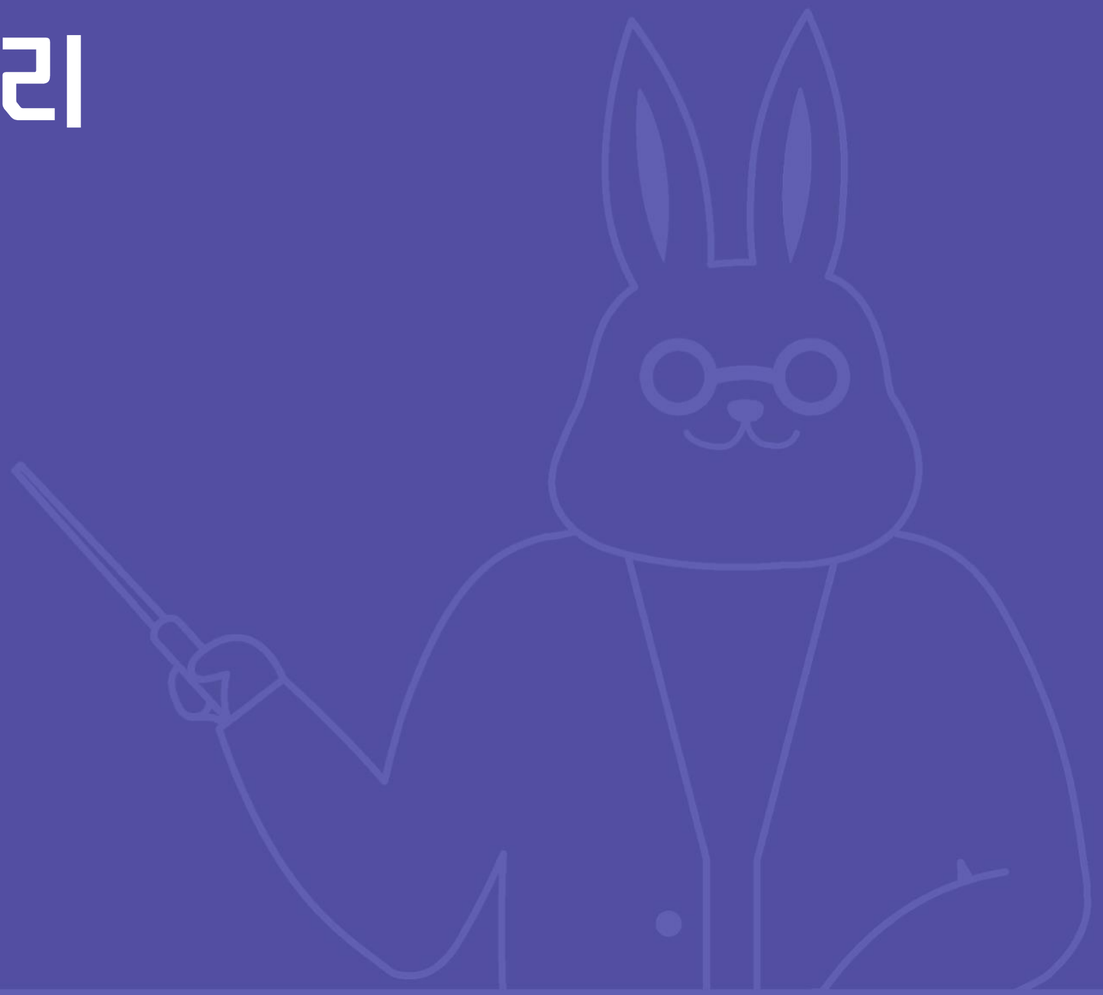
- 컴포넌트와 컴포넌트 간 상태를 공유할 때 사용.
- 부분적인 컴포넌트들의 상태 관리, 전체 앱의 상태 관리를 모두 구현.
- Context Provider 안에서 렌더링되는 컴포넌트는, useContext를 이용해 깊이 nested 된 컴포넌트라도 바로 context value를 가져옴.
- context value가 바뀌면 내부 컴포넌트는 모두 리렌더링됨.

useReducer

- useState보다 복잡한 상태를 다룰 때 사용.
- 별도의 라이브러리 없이 flux pattern에 기반한 상태 관리를 구현.
- const [state, dispatch] = useReducer(reducer, initState)
- nested state 등 복잡한 여러 개의 상태를 한꺼번에 관리하거나, 어떤 상태에 여러 가지 처리를 적용할 때 유용.
- 상태 복잡하다면, useState에 관한 callback을 내려주는 것보다 dispatch를 prop으로 내려 리렌더링을 최적화하는 것을 권장.

05

useState를 활용한 상태 관리



useState를 활용한 상태 관리

- 상위 컴포넌트에서 state와 state 변경 함수를 정의하고,
  그 state나 변경 함수를 사용하는 컴포넌트까지 prop으로 내려주는 패턴.
- state가 변경되면, 중간에 state를 넘기기만 하는 컴포넌트들도 모두
  리렌더링됨.
- 상태와 상태에 대한 변화가 단순하거나, 상대적으로 소규모 앱에서 사용하기에 적합.

useState 예시 - TodoApp

TodoApp.jsx

```
function TodoApp() {
  const [todos, setTodos] = useState([]);
  const [filter, setFilter] = useState("all");
  const [globalId, setGlobalId] = useState(3000);

  const toggleTodo = (id) => {
    setTodos((todos) =>
      todos.map((todo) =>
        todo.id === id ? { ...todo, completed:
!todo.completed } : todo
      )
    );
  };

  const deleteTodo = (id) => {
    setTodos((todos) => todos.filter((todo) =>
todo.id !== id));
  };
}
```

TodoApp.jsx

```
    const addTodo = (title) => {
      setTodos((todos) => [{ title, id: globalId + 1
  }, ...todos]);
      setGlobalId((id) => id + 1);
    };

    return <TodosPage
            state={{ todos, filter }}
            toggleTodo={toggleTodo}
            addTodo={addTodo}
            deleteTodo={deleteTodo}
            changeFilter={setFilter}
           />
  }
```

useState 예시 - TodoApp

TodosPage.jsx

```
function TodosPage({ state, addTodo, deleteTodo,
toggleTodo, changeFilter }) {

  const filteredTodos = state.todos.filter((todo)
=> {
    const { filter } = state;

    return (
      filter === "all" ||
      (filter === "completed" && todo.completed)
||
      (filter === "todo" && !todo.completed)
    );
  });
```

TodosPage.jsx

```
    return (
      <div>
        <h3>TodosPage</h3>

        <TodoForm onSubmit={addTodo} />
        <TodoFilter filter={state.filter}
  changeFilter={changeFilter} />
        <TodoList
          todos={filteredTodos}
          toggleTodo={toggleTodo}
          deleteTodo={deleteTodo}
        />
      </div>
    );
  }
```

useState 예시 - TodoApp

TodoForm.jsx

```
function TodoForm({ onSubmit }) {
  const [title, setTitle] = useState("");

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        onSubmit(title);
        setTitle("");
      }}
    >
      <label htmlFor="todo-title">Title</label>
      <input id="todo-title" type="text"
name="todo-title" onChange={(e) =>
setTitle(e.target.value)} value={title} />
      <button type="submit">Make</button>
    </form>
  );
}
```

TodoList.jsx

```
function TodoList({ todos, toggleTodo, deleteTodo
}) {
  return (
    <ul>
      {todos.map(({ title, completed, id }) => (
        <li onClick={() => toggleTodo(id)}>
          <h5>{title}</h5>
          <div>
            {completed ? "☑ " : "✎ "}
            <button onClick={() =>
deleteTodo(id)}>Delete</button>
          </div>
        </li>
      ))}
    </ul>
  );
}
```

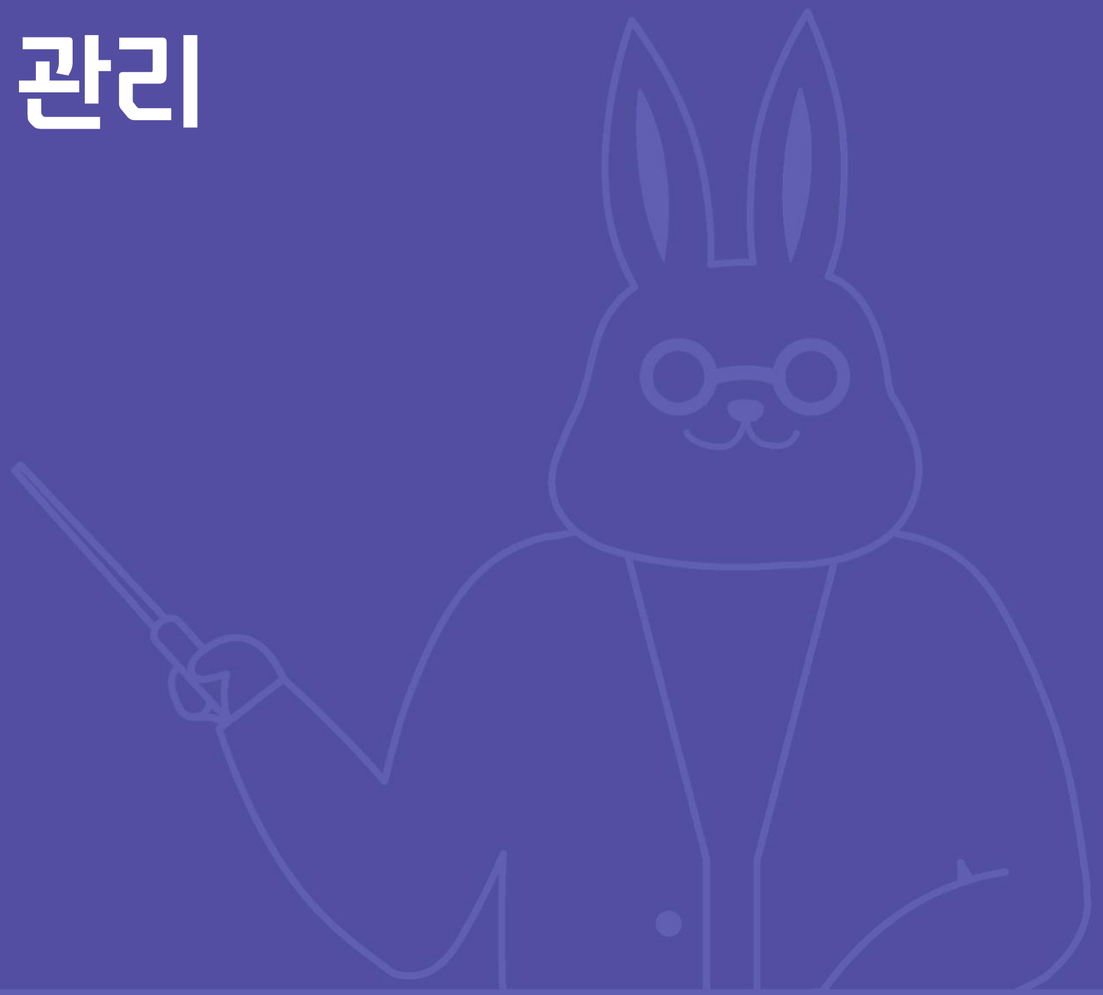
useState 예시 - TodoApp

TodoFilter.jsx

```
function TodoFilter({ filter, changeFilter }) {
  return (
    <div>
      <label htmlFor="filter">Filter</label>
      <select
        onChange={(e) => changeFilter(e.target.value)}
        id="filter"
        name="filter"
      >
        {filterList.map((filterText) => (
          <option selected={filter === filterText} value={filterText}>
            {capitalize(filterText)}
          </option>
        ))}
      </select>
    </div>
  );
}
```

06

useContext를 활용한 상태 관리



useContext를 활용한 상태 관리

- Provider 단에서 상태를 정의하고, 직접 상태와 변경 함수를 사용하는 컴포넌트에서 useContext를 이용해 바로 상태를 가져와 사용하는 패턴.
- useReducer와 함께, 복잡한 상태와 상태에 대한 변경 로직을 두 개 이상의 컴포넌트에서 활용하도록 구현 가능.
- state는 필요한 곳에서만 사용하므로, 불필요한 컴포넌트 리렌더링을 방지.
- Prop Drilling(Plumbing)을 방지하여 컴포넌트 간 결합도를 낮춤.

useState 예시 - TodoApp

TodoContext.jsx

```
const TodoContext = createContext(null);

const initialState = {
  todos: [],
  filter: "all",
  globalId: 3000,
};

function useTodoContext() {
  const context = useContext(TodoContext);
  if (!context) {
throw new Error("Use TodoContext inside
Provider.");
  }
  return context;
}

function TodoContextProvider({ children }) {
  const values = useTodoState();
  return <TodoContext.Provider
value={values}>{children}</TodoContext.Provider>;
}
```

useState 예시 - TodoApp

TodoContext.jsx

```
function reducer(state, action) {
  switch (action.type) {
    case "change.filter":
      return { ...state, filter:
action.payload.filter };
    case "init.todos":
      return { ...state, todos:
action.payload.todos };
    case "add.todo": {
      return { ...state,todos: [{ title:
action.payload.title, id: state.globalId + 1 }],
globalId: state.globalId + 1 };
    }
```

```
    case "delete.todo": {
      return { ...state, todos:
state.todos.filter((todo) => todo.id !==
action.payload.id) };
    }
    case "toggle.todo": {
      return { ...state,todos: state.todos.map((t)
=> t.id===action.payload.id ? { ...t,
completed: !t.completed } : t)};
    }
    default: return state;
  }
}
```

useState 예시 - TodoApp

TodoContext.jsx

```
function useTodoState() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const toggleTodo = useCallback( (id) => dispatch({ type: "toggle.todo", payload: { id } }), []);
  const deleteTodo = useCallback( (id) => dispatch({ type: "delete.todo", payload: { id } }), []);
  const addTodo = useCallback( (title) => dispatch({ type: "add.todo", payload: { title } }), []);
  const changeFilter = useCallback( (filter) => dispatch({ type: "change.filter", payload: { filter } }), []);
  const initializeTodos = useCallback( (todos) => dispatch({ type: "init.todos", payload: { todos } }), []);

  return { state, toggleTodo, deleteTodo, addTodo, changeFilter, initializeTodos };
}
```

useState 예시 - TodoApp

TodoApp.jsx

```jsx
function TodoApp() {
  return (
    <TodoContextProvider>
      <TodosPage />
    </TodoContextProvider>
  );
}
```

TodosPage.jsx

```jsx
function TodosPage() {
  const { initializeTodos } = useTodoContext();

  useEffect(() => {
    console.log("useEffect");
    fetchTodos().then(initializeTodos);
  }, [initializeTodos]);

  return (
    <div>
      <TodoForm />
      <TodoFilter />
      <TodoList />
    </div>
  );
}
```

useState 예시 - TodoApp

TodoForm.jsx

```jsx
function TodoForm() {
  const { addTodo } = useTodoContext();
  const [title, setTitle] = useState("");

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        addTodo(title);
        setTitle("");
      }}
    >
      <label htmlFor="todo-title">Title</label>

      <input
        id="todo-title"
        type="text"
        name="todo-title"
        onChange={(e) => setTitle(e.target.value)}
        value={title}
      />

      <button type="submit">Make</button>
    </form>
  );
}
```

TodoList.jsx

```jsx
function TodoList() {
  const { state, toggleTodo, deleteTodo } = useTodoContext();

  const { todos, filter } = state;

  const filteredTodos = todos.filter((todo) => {
    return (
      filter === "all" ||
      (filter === "completed" && todo.completed) ||
      (filter === "todo" && !todo.completed)
    );
  });

  return (
    <ul>
      {filteredTodos.map(({ title, completed, id }) => (
        <li key={id} onClick={() => toggleTodo(id)}>
          <h5>{title}</h5>
          <div>
            {completed ? "☑ " : "✎ "}
            <button onClick={() => deleteTodo(id)}>Delete</button>
          </div>
        </li>
      ))}
    </ul>
  );
}
```

useState 예시 - TodoApp

TodoForm.jsx

```jsx
function TodoForm({ onSubmit }) {
  const [title, setTitle] = useState("");

  return (
    <form
      onSubmit={(e) => {
        e.preventDefault();
        onSubmit(title);
        setTitle("");
      }}
    >
      <label htmlFor="todo-title">Title</label>
      <input id="todo-title" type="text" name="todo-title"
onChange={(e) => setTitle(e.target.value)} value={title} />
      <button type="submit">Make</button>
    </form>
  );
}
```

TodoList.jsx

```jsx
function TodoList({ todos, toggleTodo, deleteTodo }) {
  return (
    <ul>
      {todos.map(({ title, completed, id }) => (
        <li onClick={() => toggleTodo(id)}>
          <h5>{title}</h5>
          <div>
            {completed ? "☑ " : "✎ "}
            <button onClick={() =>
deleteTodo(id)}>Delete</button>
          </div>
        </li>
      ))}
    </ul>
  );
}
```

useState 예시 - TodoApp

TodoFilter.jsx

```
function TodoFilter() {
  const { state, changeFilter } =
useTodoContext();
  const { filter } = state;

  return (
    <div>
      <label htmlFor="filter">Filter</label>
      <select
        onChange={(e) =>
changeFilter(e.target.value)}
        id="filter"
        name="filter"
        value={filter}
      >
        {filterList.map((filterText) => (
          <option key={filterText}
  value={filterText}>
            {capitalize(filterText)}
          </option>
        ))}
      </select>
    </div>
  );
}
```

크레딧

/* elice */

코스 매니저
이재성

콘텐츠 제작자
김일식

강사
김일식

감수자
-

디자이너
강혜정

연락처

TEL
070-4633-2015

WEB
https://elice.io

E-MAIL
contact@elice.io