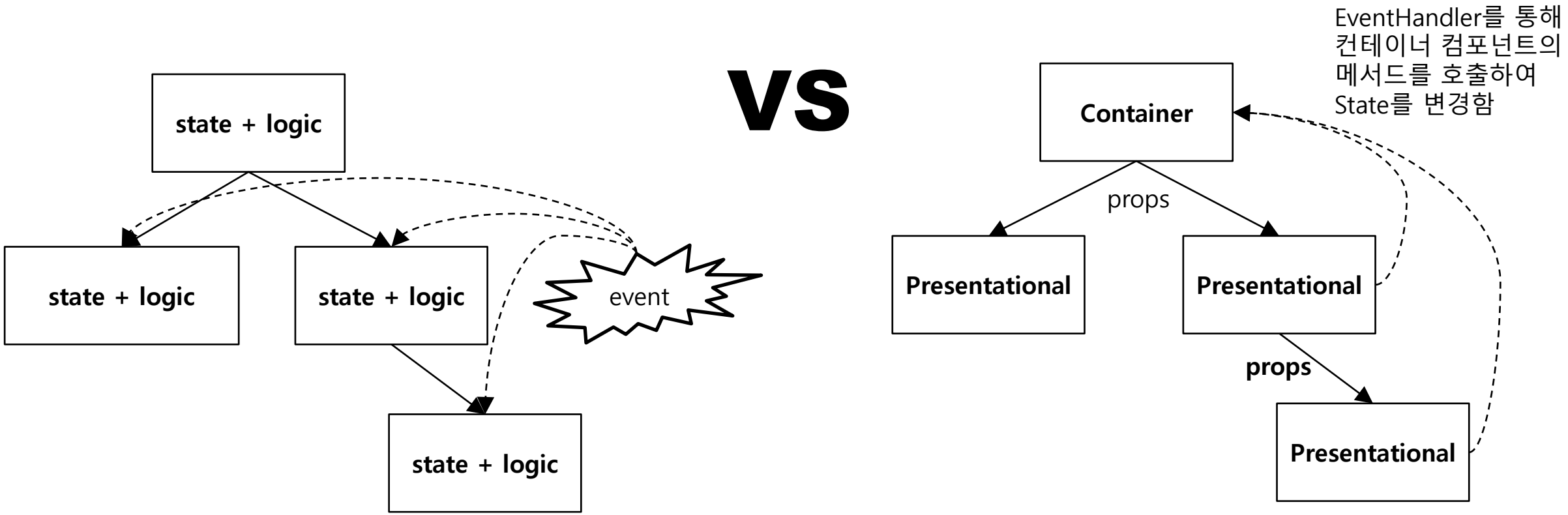
- 불변성
- react-router 심화

1. 상태와 속성 구성

❖ 상태와 속성을 어떻게 구성할 것인가?



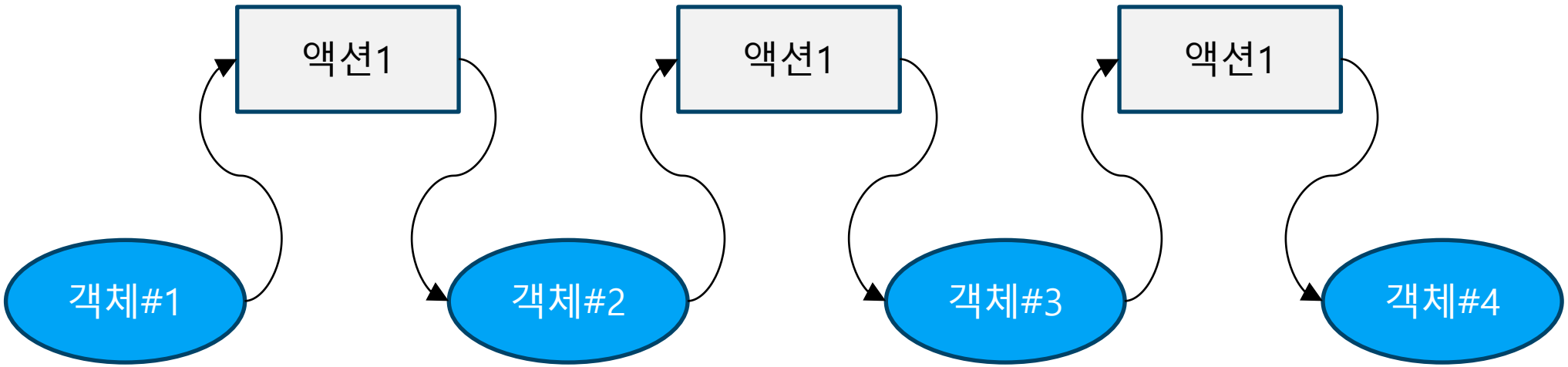
2. 불변성(Immutabiity)

❖ 불변성이 필요한 이유

- 렌더링 최적화 --〉 React.memo(), useCallback(), useMemo()
- 상태 변경 추적 --〉 Redux(immer), Zustand(immer), Recoil(immutable)

❖객체 불변성 개념

- 기존 객체를 변경하지 않고 내부 데이터, 속성이 변경된 새로운 객체를 생성

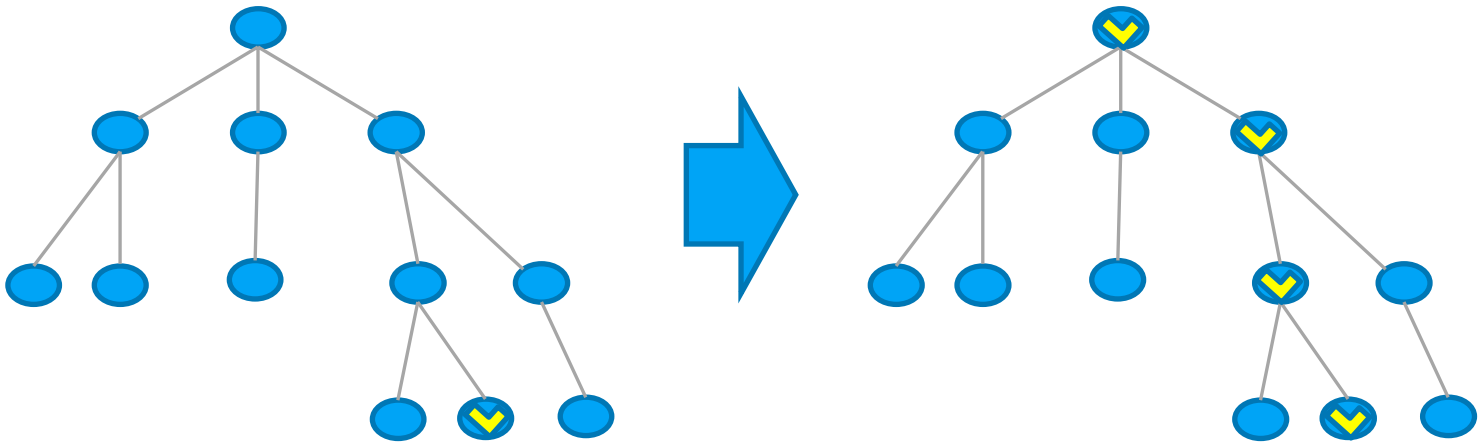


2. 불변성(Immutabiity)

❖ 렌더링 최적화에 불변성이 필요한 이유

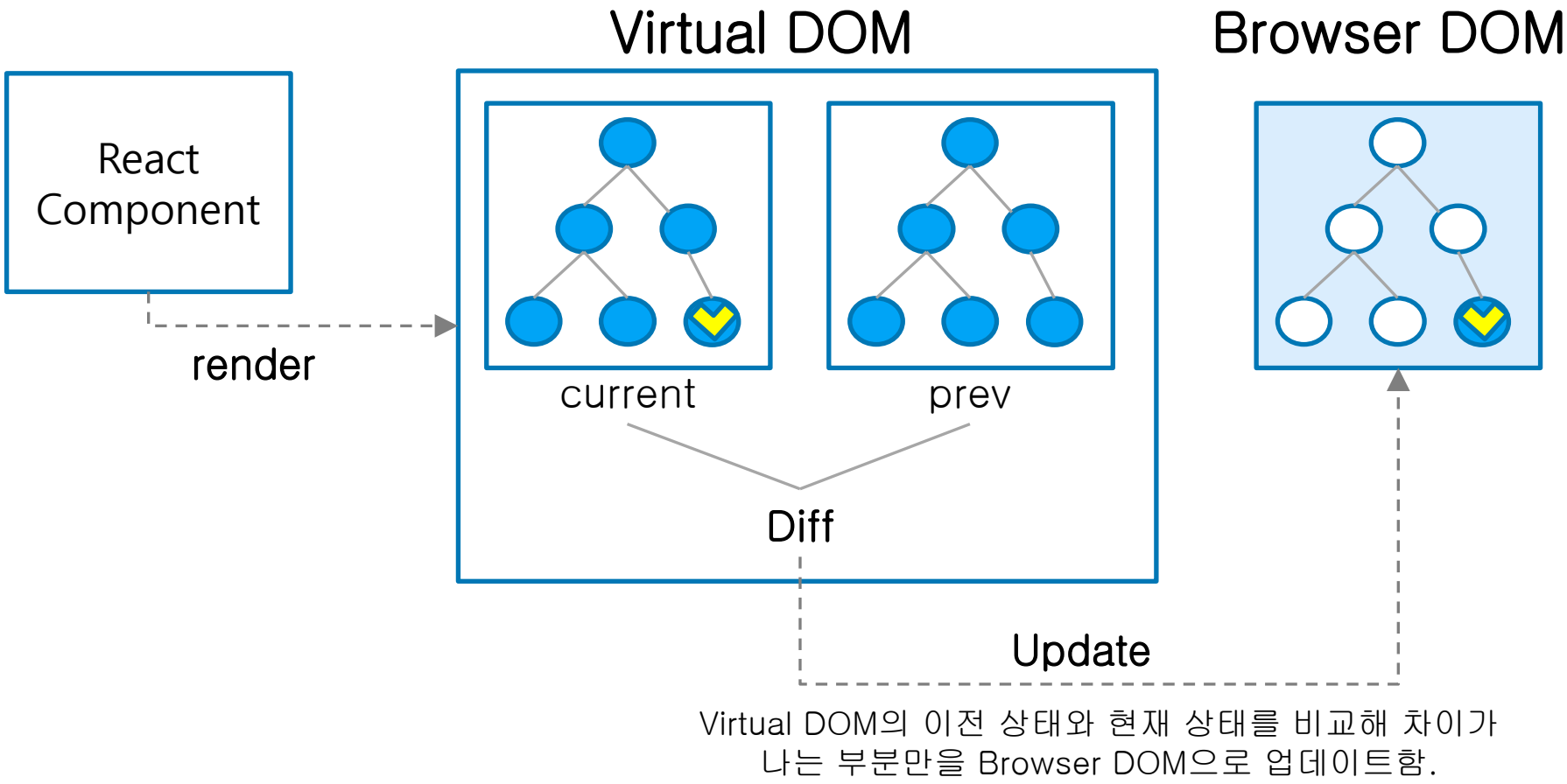
- 컴포넌트 트리에서 상태, 속성이 변경된 것이 있을 때만 가상DOM에 대한 렌더링을 해야 함
  - 이전 상태 객체와 현재 상태 객체를 비교하여 렌더링 여부 판단
  - Deep Compare(X) --〉 Shallow Compare(O)
- 불변성을 사용하지 않았다면? Shallow Compare가 불가능

❖불변성 라이브러리가 변경하는 방법



2. 불변성(Immutabiity)

❖ Virtual DOM의 작동 방식 리뷰



2. 불변성(Immutabiity)

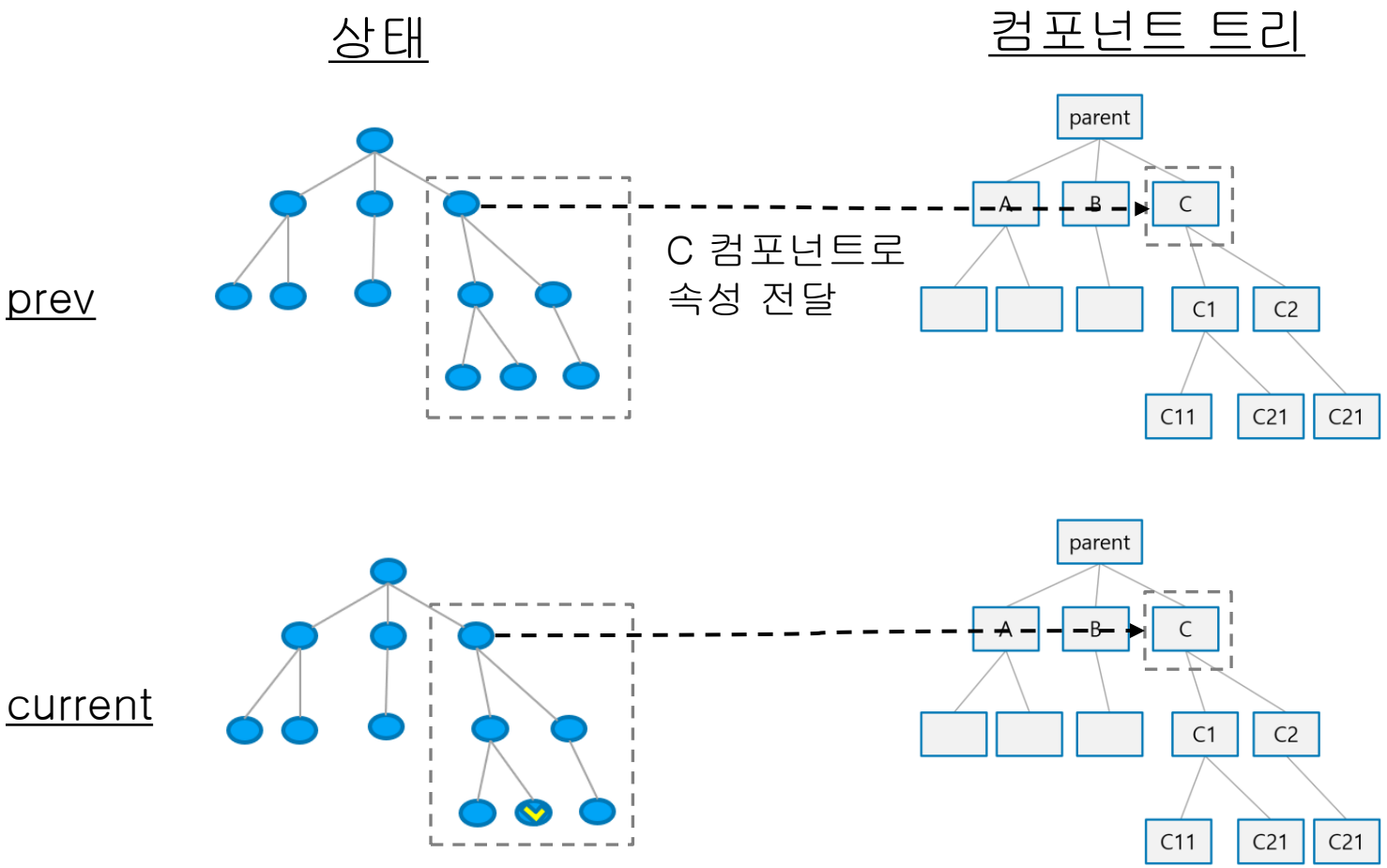
❖ 상태와 렌더링

- React 앱에서 상태(state)가 변경되면 UI가 렌더링됨
- 상태를 변경하기 위해 setter 메서드를 사용함
  - const [x, **setX**] = useState(0);
  - **setter 메서드는 상태만 변경하는 것이 아니라 React 앱에게 알려 상태를 보유한 컴포넌트가 re-render되게 함**
- 부모 컴포넌트에서 re-render가 일어나면 포함된 모든 자식 컴포넌트는 re-render!!
  - 자식 컴포넌트가 사용하는 데이터(특히 props)에 변경된 것이 없다하더라도...
  - 렌더링 최적화는 자식 컴포넌트 중 props가 동일하다면 re-render가 일어나지 않도록 하는 것
  - 하지만 상태 데이터가 객체라면 이것이 쉽지 않음
- Virtual DOM
  - 이전 렌더링 결과와 현재의 렌더링 결과를 비교해 브라우저 DOM에 업데이트함
  - Virtual DOM에 불필요하게 re-render 하는 것도 성능에 영향을 줌.
    - 속성이나 상태가 바뀐게 없다면 re-rende를 하지 않도록 하여 Virtual DOM에 대한 쓰기조차 하지 않도록...

2. 불변성(Immutabiity)

❖ Deep Compare

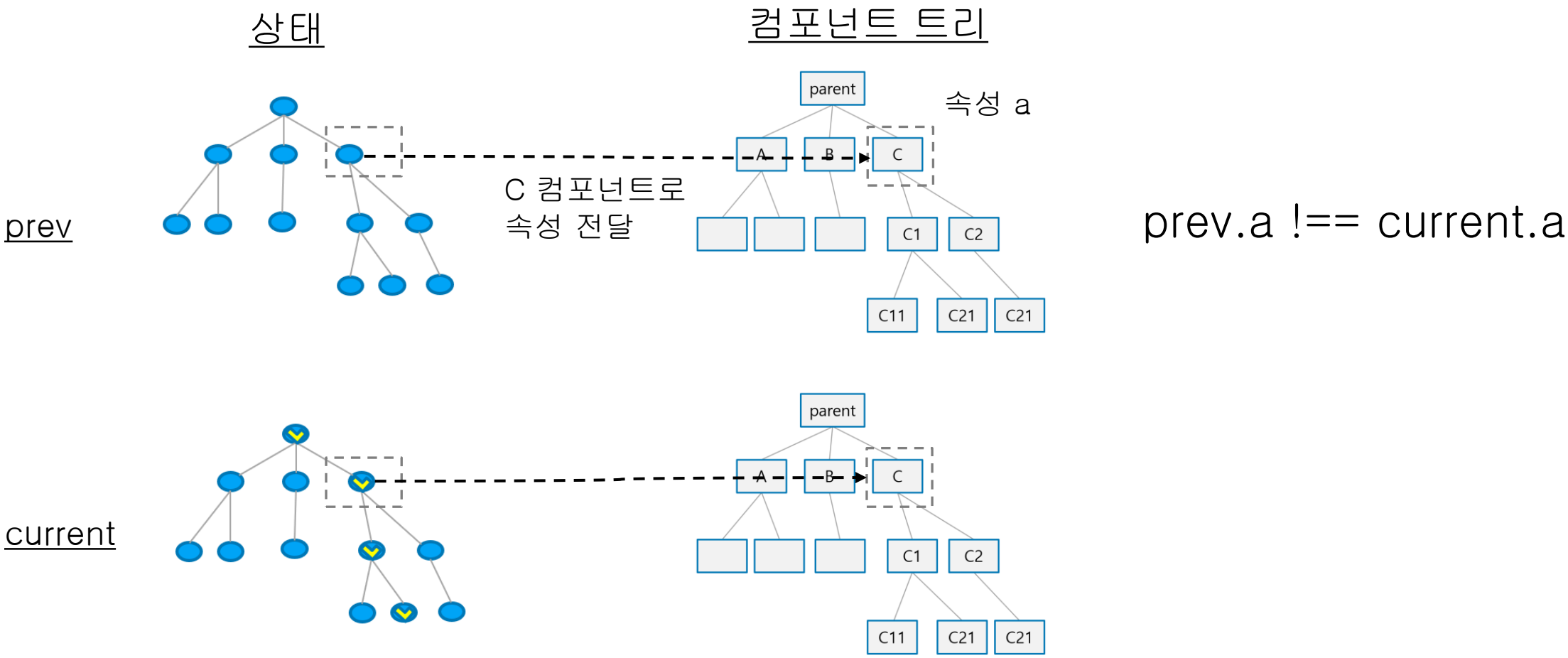
- re-render할지 여부를 결정하기 위해 객체의 트리를 따라 내려가면서 모두 비교함 --〉성능 저하



2. 불변성(Immutabiity)

❖ Shallow Compare

- re-render할지 여부를 결정하기 속성으로 전달받은 객체의 메모리 주소만 비교함



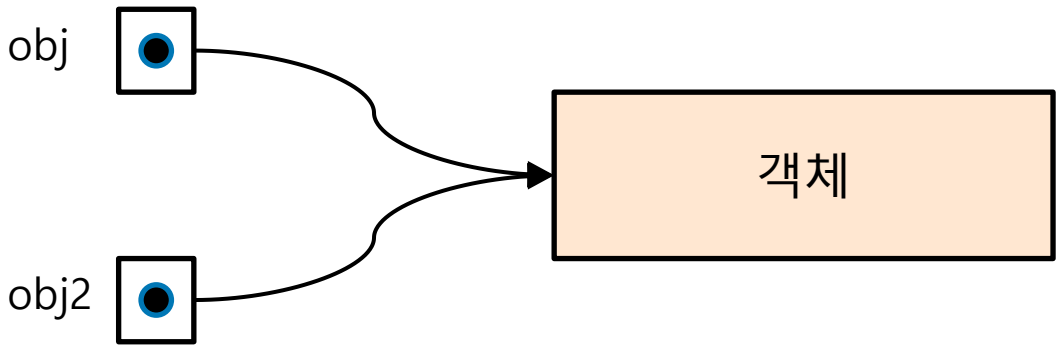
2. 불변성(Immutabiity)

❖불변성과 shallow copy

- 다음 코드는 불변성을 가지는가?

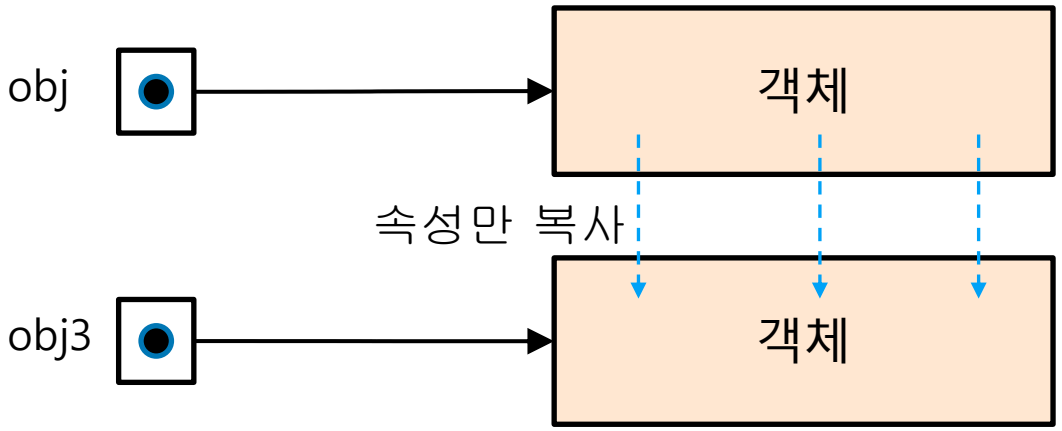
```
let obj2 = obj;                 //shallow copy
obj2.name = "이순신";
```

- shallow copy 이므로 같은 객체를 참조함
  - 객체의 메모리 주소를 복사
  - 원본이 함께 변경되므로 불변성(X)



❖불변성을 위해 전개(spread) 연산자 사용

```
//기존 객체의 속성값을 복사한 후 name 속성을 이순신으로 변경한
새로운 객체를 생성함.
//따라서 obj3의 속성을 변경한다하더라도 obj의 값은 변경되지 않음
let obj3 = { ...obj, name: "이순신" };
```



❖하지만 전개 연산자는 복잡한 트리구조의 객체는 불변성을 제공하지 않음

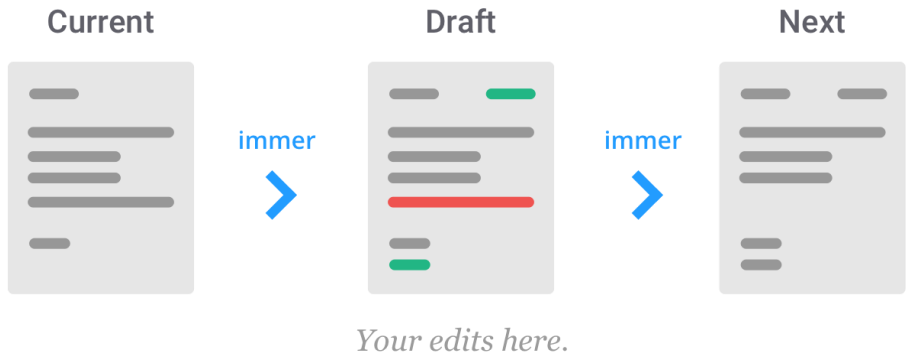
- 전개연산자를 사용한 수준의 속성까지만 불변성을 제공함

3. immer 불변성 라이브러리

❖복잡한 트리구조에 대한 불변성

- immer 라이브러리 사용
  - npm install immer
- immer 사용
  - 쉬운 불변성 확보 방법 제공.
  - 자바스크립트 객체, 배열의 접근방식을 그대로 사용함
  - 객체 트리 끝단의 값을 변경하면 Root로 올라가능 경로 상의 객체를 모두 새로운 것으로 생성함.

```
//immer 9버전용
import produce from "immer"
//immer 10버전은 다음과 같이
import { produce } from "immer"

const currentState = [
    {  todo: "Learn es6",  done: true   },
    {  todo: "Try immer",   done: false  }
]

const nextState = produce(currentState, (draft) => {
    draft[1].done = true
})
```



3. immer 불변성 라이브러리

❖ 객체의 불변성(immutability)이란?

- '변경'은 기존 객체를 변경하는 것이 아니라 새로운 버전의 객체를 만듬
- 복잡한 객체 예
  - quiz 객체 : 개발시에 만날 수도 있는 복잡도
  - obj 객체 : 단순한 객체
- 이 객체를 이용해 불변성이 무엇인지 살펴봄

❖ 예제 준비

- https://stackblitz.com/ 에서 React(ES6) 템플릿 사용
- dependencies 에 immer 추가
- index.js에 우측의 객체 추가

```
let quiz = {
   "students" : ["홍길동", "성춘향", "박문수", "변학도" ],
   "quizlist": [
    {
      "question": "한국 프로야구 팀이 아닌것은?",
      "options": [
         { "no":1, "option":"삼성라이온스" },
         { "no":2, "option":"기아타이거스" },
         { "no":3, "option":"두산베어스" },
         { "no":4, "option":"LA다져스" }
      ],
      "answer": 4
    },
    {
      "question": "2018년 크리스마스는 무슨 요일인가?",
      "options": [
         { "no":1, "option":"월" },
         { "no":2, "option":"화" },
         { "no":3, "option":"수" },
         { "no":4, "option":"목" }
      ],
      "answer": 2
    }
   ]
}
```

3. immer 불변성 라이브러리

- immer 테스트

```
……
import { produce } from "immer";
……
let quiz = {  ……  };

const quiz2 = produce(quiz, draft => {
  draft.quizlist[0].options[0].option = "LG트윈스";
});

console.log(quiz.quizlist[0].options[0].option);

//false, false, false, false, false, true
console.log(quiz === quiz2)
console.log(quiz.quizlist === quiz2.quizlist)
console.log(quiz.quizlist[0] === quiz2.quizlist[0])
console.log(quiz.quizlist[0].options[0] === quiz2.quizlist[0].options[0])
console.log(quiz.quizlist[0].options[0].option === quiz2.quizlist[0].options[0].option)
console.log(quiz.students === quiz2.students)
```

- 변경된 객체의 끝단의 값으로부터 거슬러 올라가는 경로상의 객체는 모두 새로운 것으로 변경
  - 경로상이 아닌 것은 기존 객체를 유지

3. immer 불변성 라이브러리

- 실행 결과

index.js

```
2  import { createRoot } from 'react-dom/client';
3  import { produce } from 'immer';
4  import App from './App';
5
6  const root = createRoot(document.getElementById('app'));
7
8  root.render(···
12 );
13
14 let quiz = {···
38 };
39
40 const quiz2 = produce(quiz, (draft) => {
41   draft.quizlist[0].options[0].option = 'LG트윈스';
42 });
43
44 console.log(quiz.quizlist[0].options[0].option);
45
46 //false, false, false, false, false, true
47 console.log(quiz === quiz2);
48 console.log(quiz.quizlist === quiz2.quizlist);
49 console.log(quiz.quizlist[0] === quiz2.quizlist[0]);
50 console.log(quiz.quizlist[0].options[0] === quiz2.quizlist[0].options[0]);
51 console.log(
52   quiz.quizlist[0].options[0].option === quiz2.quizlist[0].options[0].option
53 );
54 console.log(quiz.students === quiz2.students);
```

stackblitz-starters-untear.sta...

Hello StackBlitz!

Start editing to see some magic happen :)

Console 7

Preview (local) Clear on reload

```
Console was cleared
삼성라이온스
false
false
false
false
false
true
```

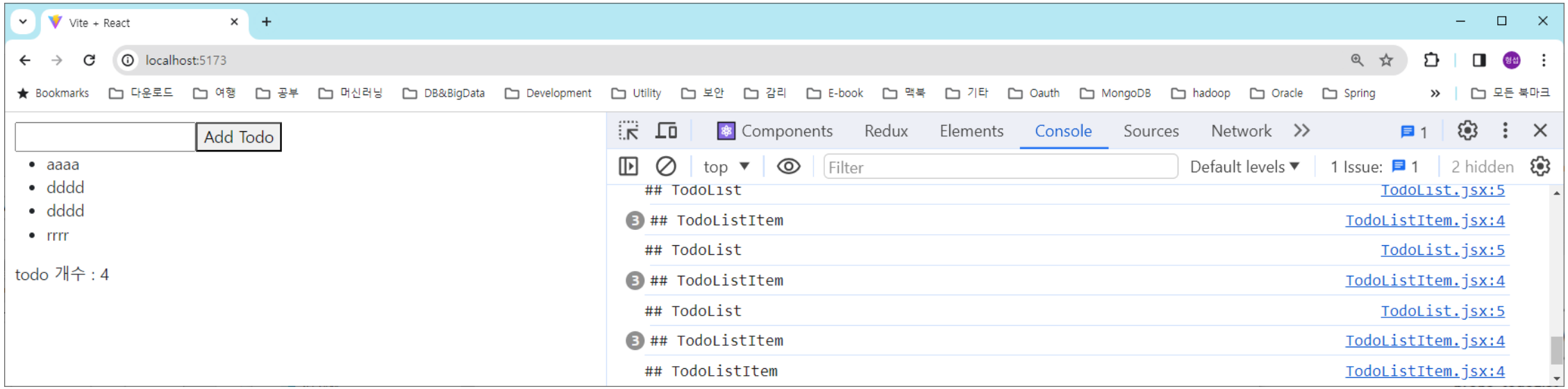
3. immer 불변성 라이브러리

❖immer를 반드시 사용해야 하는가?

- 그렇지 않다. 하지만 바람직함.
  - 간단한 객체는 Spread 연산자(...)을 이용할 수 있음
- 특히 UI 렌더링 성능 최적화를 위해서는 반드시 필요함.
- 상태 데이터에 대해 불변성을 확보하는 것이 중요하다는 점을 인식해야 함

4. 렌더링 최적화

- ❖ React.memo()와 useCallback() 훅 활용
  - ▪ 불변성을 이용한 상태 변경은 필수
- ❖ 최적화 이전의 todolist-app 예제 확인
  - ▪ 강사가 제공하는 예제를 확인하고 실행해 봄
    - todo를 추가할 때마다 모든 TodoListItem이 렌더링됨.
    - 심지어 새로운 todo 입력을 위해 타이핑할때도 모두 렌더링
      - App 컴포넌트의 상태가 바뀌기 때문에...



4.1 최적화 1단계

❖ React.memo() 고차 함수 이용

- 컴포넌트의 렌더링 결과를 캐싱함
- 상태나 속성이 변경된 것이 없다면 렌더링하지 않고 메모이징된 캐시를 사용함
  - Shallow Compare 이용 확인
- 사용 방법

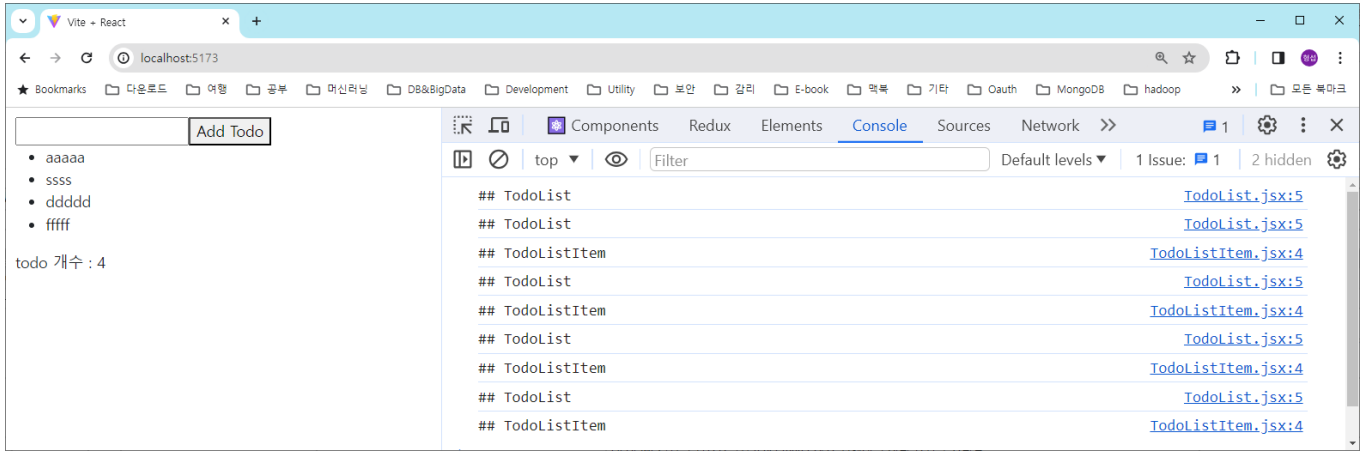
```
export default React.memo(기존 컴포넌트);
```

❖React.memo() 적용

- TodoList, TodoListItem에 React.memo() 고차함수를 적용 후 todo를 계속해서 추가해 봄
- 실행 결과
  - 타이핑할 때 다시 렌더링되지 않음
  - todo를 추가한 경우 추가된 것만 렌더링됨.

4.1 최적화 1단계

❖실행 결과 : 최적화된 듯한 느낌!



- 하지만 TodoList, TodoListItem에 삭제 기능을 추가하면 다시 모두 렌더링됨
  - App, TodoList 컴포넌트에서 각각 TodoList, TodoListItem으로 속성을 전달하도록 변경

```
<TodoList key={todoItem.id} todoListItem={todoItem} deleteTodo={deleteTodo} />
<TodoListItem key={todoItem.id} todoListItem={todoItem} deleteTodo={props.deleteTodo} />
```

  - TodoListItem 컴포넌트에 삭제 버튼 추가

```
return <li>{props.todoListItem.todo}
        <button onClick={()=>props.deleteTodo(props.todoListItem.id)}>삭제</button></li>;
```

4.2 최적화 2단계

❖ 삭제 기능을 추가한 후 왜 다시 모두 렌더링되는가?

- 렌더링 과정 확인
  - AppContainer 컴포넌트의 상태가 바뀜
  - AppContainer 컴포넌트 내부의 함수가 다시 생성됨(deleteTodo 포함)
  - 새롭게 생성된 deleteTodo 함수가 속성으로 App – TodoList – TodoListItem 으로 전달
  - 속성(deleteTodo)이 이전과 비교(Shallow Compare)하여 다르므로 TodoListItem 다시 렌더링
- 원인은 AppContainer 의 deleteTodo 함수가 매번 새롭게 생성되기 때문임
- 이 문제를 해결하려면
  - useCallback() 훅

❖ useCallback() 훅

- 컴포넌트 내부의 함수를 캐싱함

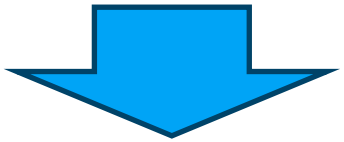
```
//의존 배열 객체가 변경되면 메모이징된 함수를 새롭게 생성함
const memoizedFunction = useCallback(기존 함수, 의존 배열 객체);
```

4.2 최적화 2단계

❖ 기존 예제에 useCallback 훅 적용

- App 컴포넌트의 addTodo, deleteTodo 함수를 다음과 같은 형태로 변경

```
const deleteTodo = (id)=> {
  let newTodoList = todoList.filter((item)=>item.id !== id);
  setTodoList(newTodoList);
}
```



```
const deleteTodo = useCallback((id)=> {
  let newTodoList = todoList.filter((item)=>item.id !== id);
  setTodoList(newTodoList);
}, [todoList]);
```

4.2 최적화 2단계

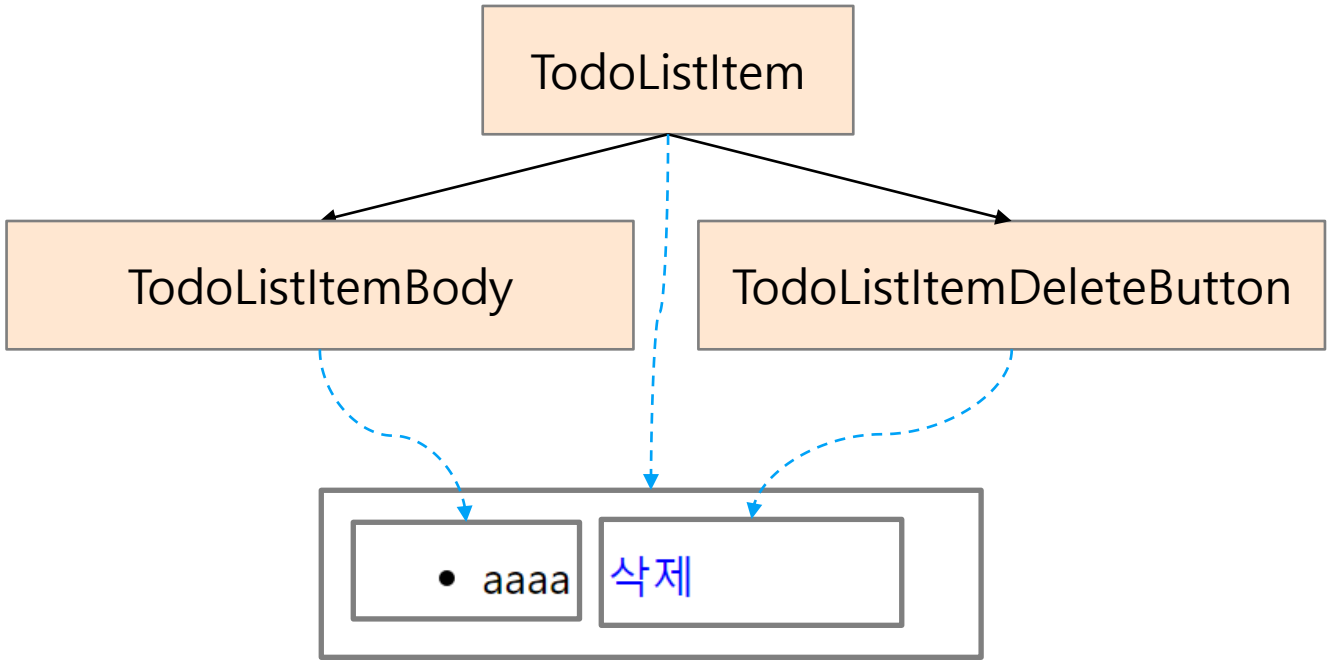
❖ useCallback() 훅 적용결과

- 새로운 todo를 타이핑할 때는 다시 렌더링되지 않음
- 목록의 todo를 삭제할때 모두 렌더링
- 새로운 todo가 todoList 상태에 추가되면 모두 렌더링
  - 이유 : useCallback의 의존 객체인 todoList가 변경되면 deleteTodo 함수가 다시 생성되기 때문에
  - 이 때의 re-render는 꼭 필요한 것임
    - useCallback는 메서드가 생성될 때 의존 객체 값을 사용하기 때문에...

4.3 렌더링 최적화 3단계

❖근본적인 해결책은?

- 바뀌는 속성과 바뀌지 않는 속성을 구분하여 컴포넌트를 분할하라
  - TodoListItemBody는 다시 렌더링되지 않음
  - TodoListItemDeleteButton은 다시 렌더링됨
- Application 수준의 상태관리를 사용하고 액션 메서드를 자식 컴포넌트에 직접 바인딩하라.



5. react-router의 중첩 라우트

❖ 중첩 라우트

- 〈Route /〉 에 렌더링된 컴포넌트에 기존 라우트의 중첩된 〈Route /〉의 컴포넌트가 나타나도록 구성하는 〈Route /〉 컴포넌트의 적용방법

❖ 기존 제공 예제 Route : 중첩라우트 아님

```
<Routes>
  ……
  <Route path="/songs" element={<SongList songs={songs} />} />
  <Route path="/songs/:id" element={<SongDetail songs={songs} />} />
</Routes>
```

- /songs 로 요청하는 경우 : SongList 컴포넌트 렌더링
- /songs/:id 로 요청하는 경우 : SongDetail 컴포넌트 렌더링

5. react-router의 중첩 라우트

❖ 중첩라우트

```
<Routes>
  ……
  <Route path="/songs" element={<SongList songs={songs} />}>
    <Route path=":id" element={<Player songs={songs} />} />
  </Route>
</Routes>
```

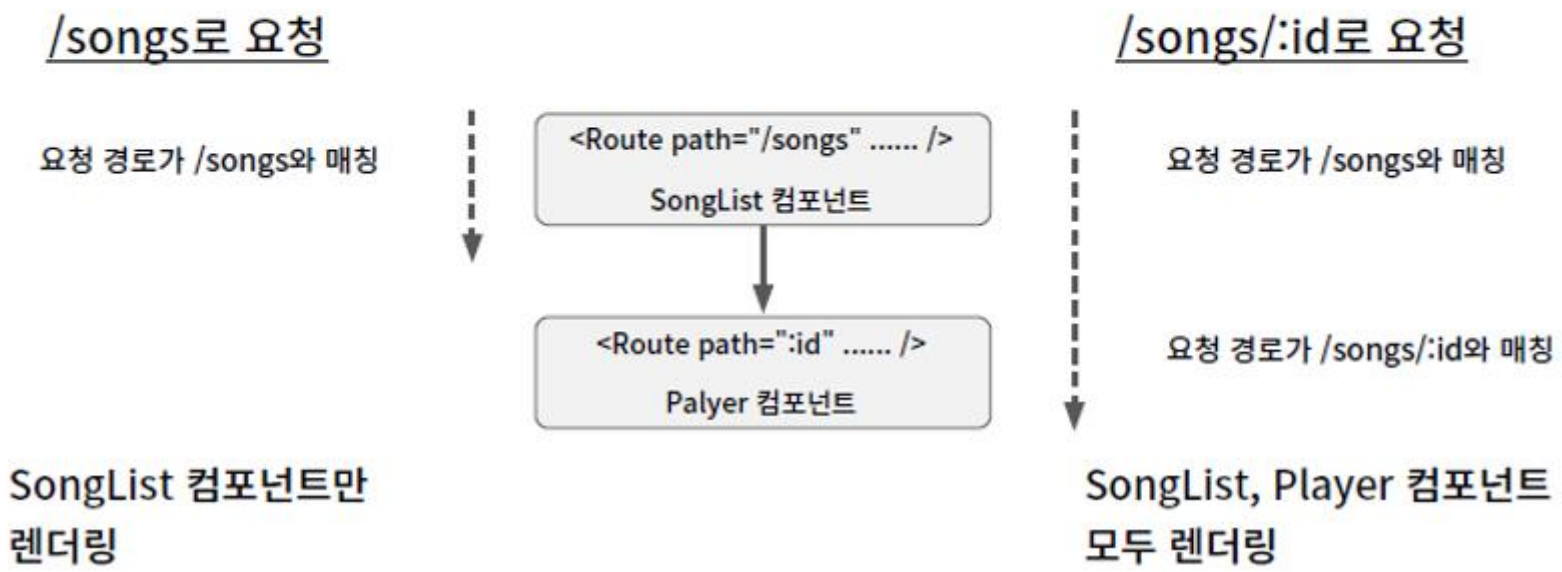
- /songs 로 요청하는 경우 : SongList 컴포넌트 렌더링
- /songs/:id 로 요청하는 경우 : SongList, Player 컴포넌트 렌더링
- 상위 라우트에 의해 렌더링되는 컴포넌트에 〈Outlet /〉 컴포넌트가 존재해야 함

```
//SongList 컴포넌트에…
<div>
   <h2 className="m-5">Song List</h2>
   <ul className="list-group">{list}</ul>
   <Outlet />
</div>
```

5. react-router의 중첩 라우트

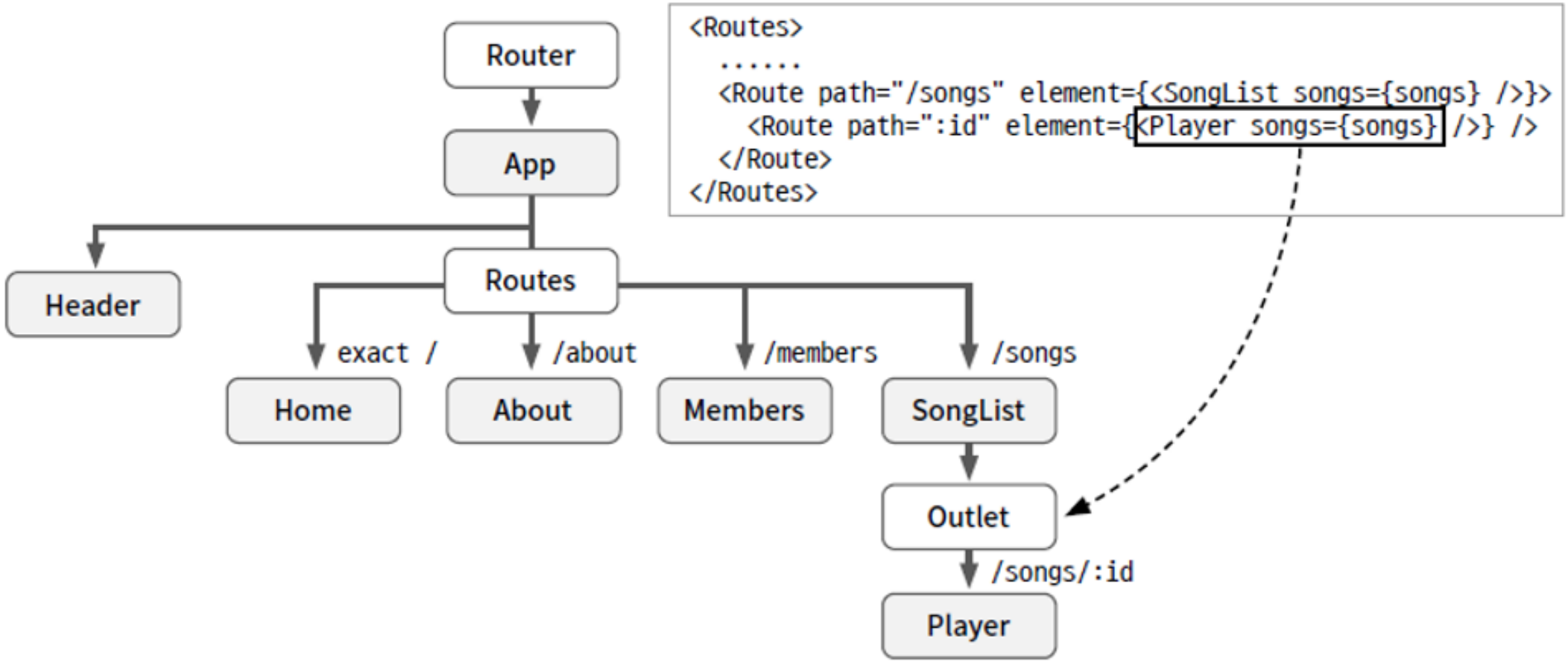
❖ 실행 방식

```
......
  <Route path="/songs" element={<SongList songs={songs} />}>
     <Route path=":id" element={<Player songs={songs} />} />
  </Route>
  ......
```



5. react-router의 중첩 라우트

❖ 중첩 라우트 실행 구조



5. react-router의 중첩 라우트

❖ 중첩 인덱스 라우트

- 기본 자식 라우트의 성격(default child route)
- 요청된 경로가 부모 경로와 일치할 때 기본적으로 보여줄 자식 라우트

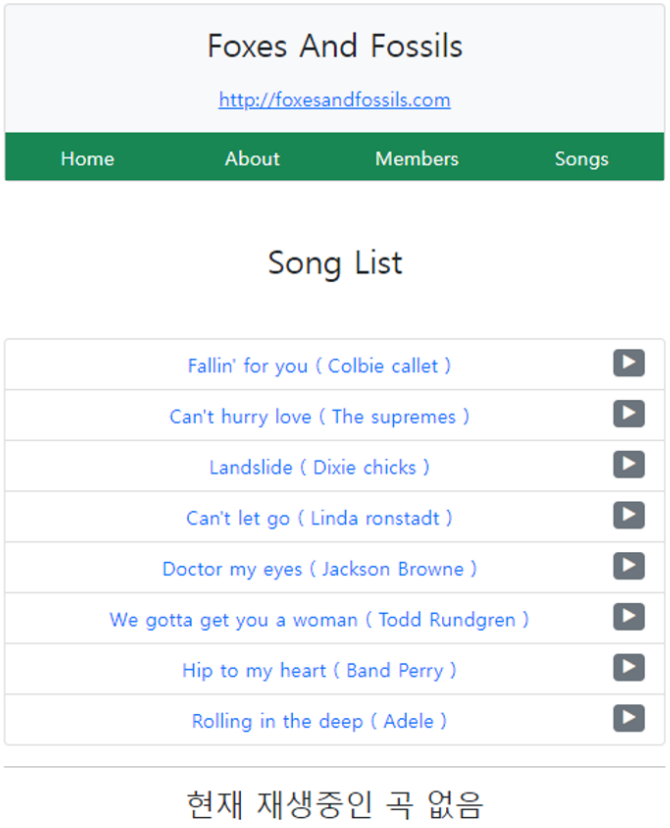
```
<Routes>
  ……
  <Route path="/songs" element={<SongList songs={songs} />}>
    <Route index element={<SongIndex />} />
    <Route path=":id" element={<Player songs={songs} />} />
  </Route>
</Routes>
```

- /songs 로 요청할 경우 : SongList, SongIndex 컴포넌트 렌더링
- /songs/:id 로 요청한 경우 : SongList, Player 컴포넌트 렌더링

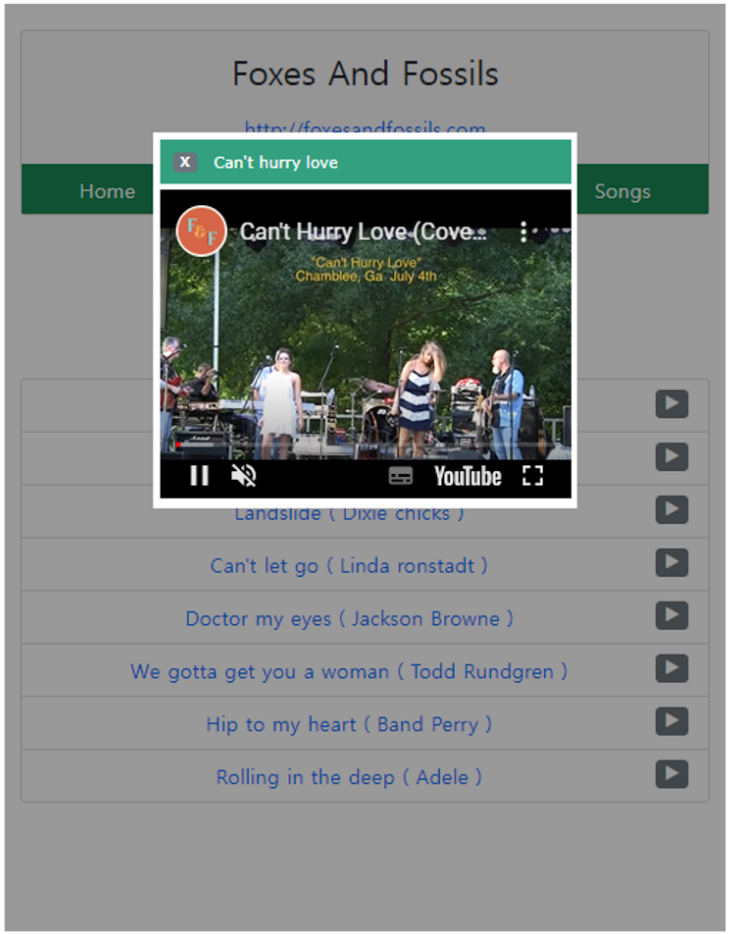
5. react-router의 중첩 라우트

❖ 중첩 인덱스 라우트

/songs 요청



/songs/:id 요청



6. reacr-router가 제공하는 훅

❖react-router가 제공하는 hook

- useMatch(경로패턴)
  - 현재 요청 경로가 지정한 경로 패턴에 부합하는 경우 PathMatch 객체를 리턴함
- useParams()
  - URI 경로 파라미터 값을 포함한 Params 객체를 리턴함
  - 이미 URI 파라미터 처리 방법을 학습할 때 살펴보았음
- useSearchParams()
  - 현재 요청의 쿼리문자열을 읽거나 수정할 수 있음
- useLocation()
  - 현재 요청된 경로 정보를 포함하는 Location 객체를 리턴함
- useNavigate()
  - 화면 전환(이동)을 위한 Navigate() 함수를 리턴함
- useOutletContext()
  - 상위 경로에 상태를 저장하고 자식 경로에 대한 Outlet에 렌더링하는 자식 컴포넌트에서 상태를 이용할 수 있도록 함

6. reacr-router가 제공하는 훅

❖useMatch()

- 현재 경로를 경로 패턴에 부합되는지 매칭하고 PathMatch 경로를 리턴함
- 사용 방법

```
//useMatch 훅의 인자로 <Route/>의 path 속성에 지정하던 경로 패턴을 전달합니다.
//  ex) /songs/:id
//경로가 일치하면 매칭된 경로 정보를 리턴하며 일치하지 않으면 null을 리턴함
const PathMatch = useMatch(경로 패턴)
```

- PathMatch 객체의 속성
  - params : URI 경로의 파라미터, ex) params.id
  - pathname : 요청된 경로, ex) /songs/2
  - pattern : 요청된 경로 패턴, ex) /songs/:id

6. reacr-router가 제공하는 훅

❖useMatch 훅을 이용한 예제

- 플레이중인 유튜브 영상의 항목을 조금 다른 스타일로 보여줌
- SongList 컴포넌트에서...

```
import { Link, Outlet, useMatch } from "react-router-dom";

const SongList = (props) => {
  const pathMatch = useMatch("/songs/:id");
  let param_id = pathMatch?.params.id ? parseInt(pathMatch.params.id, 10) : -1;

  let list = props.songs.map((song) => {
    return (
      <li key={song.id} className={param_id === song.id ? "list-group-item list-group-item-secondary" : "list-group-item"}>
        <Link to={`/songs/${song.id}`} style={{ textDecoration: "none" }}>
          {song.title} ( {song.musician} )
          <span className="float-end badge bg-secondary">
            <i className="fa fa-play"></i>
          </span>
        </Link>
      </li>
    );
  });
  .....(생략)
};
```

6. reacr-router가 제공하는 훅

❖useOutletContext 훅

- 중첩된 라우트를 사용할 때 상위 경로의 〈Outlet /〉을 이용해 Context 정보를 저장해두고 하위 경로에서 접근할 수 있도록 함.
- 사용방법
  - 상태 데이터 또는 속성을 〈Outlet /〉의 context에 추가함.
  - 중첩 라우트의 자식 경로 컴포넌트에서 useOutletContext()를 이용해 접근자 획득

〈 Route 최상위 컴포넌트에서 〉

```
const ParentComponent = () => {
   const [title, setTitle] =
       useState('Hello React!!');
   return (
      ……
     <Outlet context={ { title } } />
      ……
   )
};
export default ParentComponent;
```

〈중첩 Route 하위 경로의 Component 에서〉

```
const ChildComponent = () => {
  const { title } = useOutletContext ();
  ……
}
```

6. reacr-router가 제공하는 훅

❖useOutletContext 훅 적용

- App.jsx에서 Route 변경

```
<Route path="/songs" element={<SongList songs={songs} />}>
  <Route index element={<SongIndex />} />
  <Route path=":id" element={<Player />} />
</Route>
```

- SongList 컴포넌트의 Outlet에 Context 추가

```
……
<div>
    ……
    <Outlet context={{ songs: props.songs }}/>
</div>
……
```

6. reacr-router가 제공하는 훅

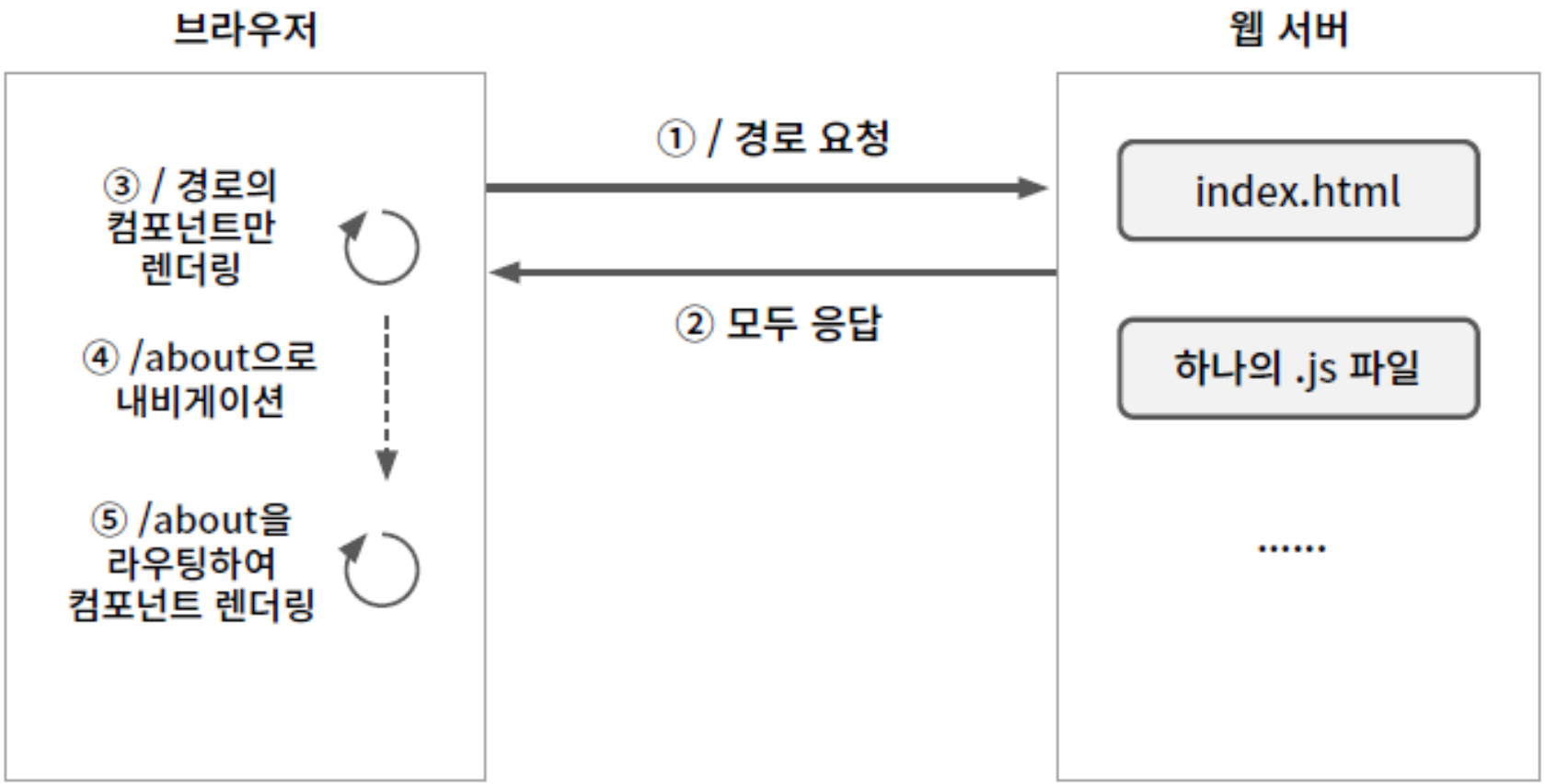
- Player 컴포넌트에서 useContext 훅 사용
  - 속성이 아닌 useOutletContext()를 이용해 songs 데이터 접근

```
……
import { useParams, useNavigate, useOutletContext } from "react-router";
……
const Player = () => {
  const { songs } = useOutletContext();
  const params = useParams();
  const navigate = useNavigate();
  const [title, setTitle] = useState("");
  const [youtubeLink, setYoutubeLink] = useState("");

  useEffect(() => {
    const id = params.id ? parseInt(params.id, 10) : 0;
    //const song = props.songs.find((song) => song.id === id);
    const song = songs.find((song) => song.id === id);
    ……(생략)
  });
  return (
    ……(생략)
  );
};
export default Player;
```

7. Lazy Loading

❖ 대규모 SPA 앱에서 첫 화면의 로딩 속도가 느린 이유

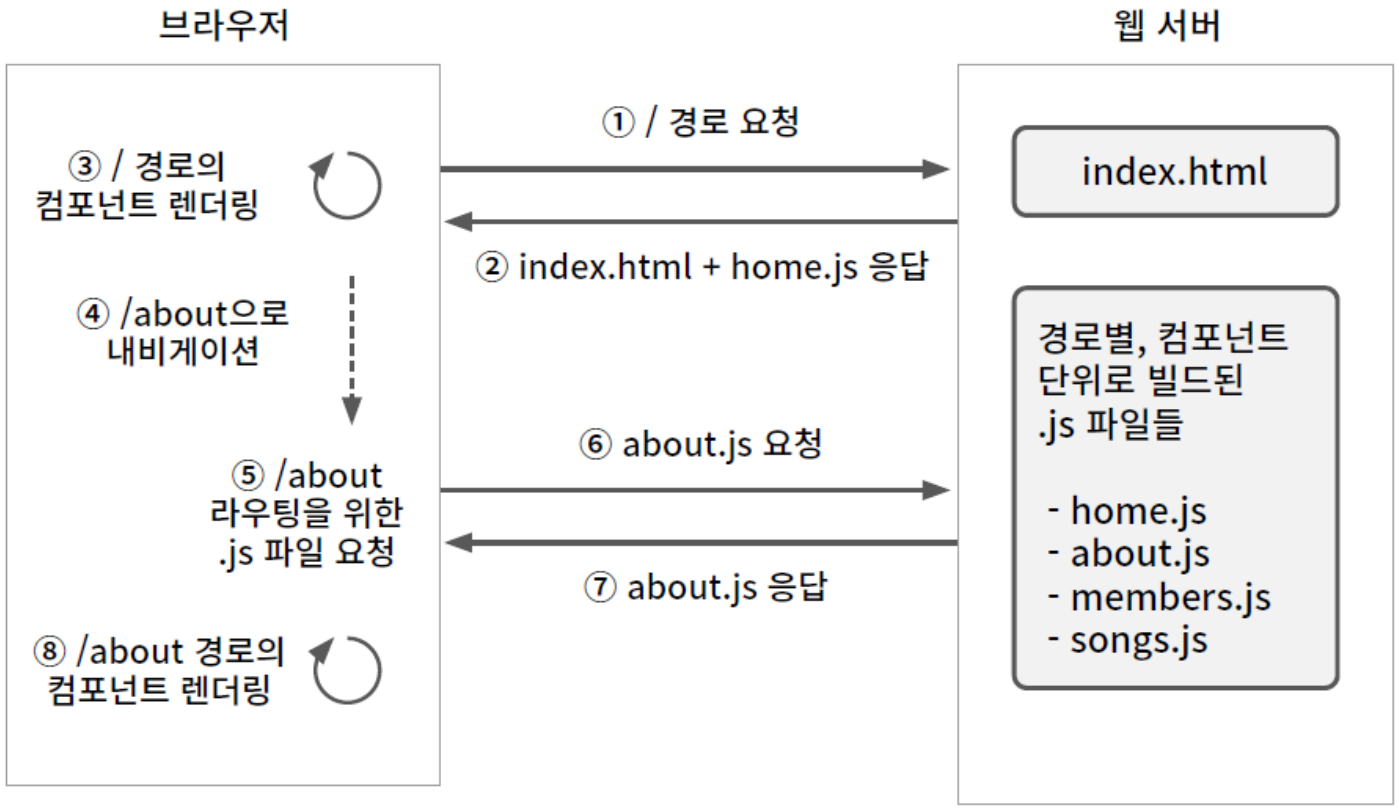
- 요청, 응답 방식



7.1 Lazy Loading

❖ Lazy Loading의 의미

- 리액트 애플리케이션의 수많은 화면과 컴포넌트 코드를 적절히 구분하여 화면, 그룹 단위로 여러개의 파일로 빌드함
  - 이 여러개의 파일을 청크(Chunk)라고 부름
- 브라우저에서 컴포넌트가 필요한 시점에 서버에 요청해 청크를 받아온 후 렌더링하는 방법



7.1 Lazy Loading

❖ 적용 방법

```
//기존의 정적 import 방법
import Home from './Home '        //1번

//React.lazy()와 import 함수 사용
const Home = React.lazy(() => import("./Home"));  //2번

//webpackChunkName 지정,  3번
const Home = React.lazy(()=> import(/* webpackChunkName:"home" */ "./Home"));
const Blog = React.lazy(()=> import(/* webpackChunkName:"home" */ "./Blog"));
```

7.1 Lazy Loading

Note

webpackChunkName을 사용하려면 알아둬야 할 사항

webpackChunkName은 이름에서 알 수 있듯이 webpack이라는 빌드 시스템이 지원하는 가능입니다. CRA(create-react-app) 도구를 이용해 리액트 프로젝트를 생성했다면 webpack이 기본으로 지원되므로 별도의 설정 없이 webpackChunkName 가능을 사용할 수 있지만, Vite로 생성된 프로젝트인 경우는 별도의 설정이 필요합니다. 설정 방법은 다음과 같습니다.

1. Vite에서 사용할 수 있도록 다음 명령어를 이용해 webpackChunkName 플러그인을 설치합니다.

```
> npm install -D vite-plugin-webpackchunkname
```

2. vite.config.js에 다음과 같은 vite-plugin-webpackchunkname 플러그인의 설정을 추가합니다.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import { manualChunksPlugin } from 'vite-plugin-webpackchunkname'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [ react(), manualChunksPlugin() ],
});
```

7.2 Suspense

❖청크파일을 웹서버에서 받아올 때 지연시간이 발생하면?

- 사용자에게 로딩중임을 나타내는 화면을 보여주어야 함
  - Spinner UI , Fallback UI
- 이 기능을 손쉽게 구현할 수 있도록 도와주는 컴포넌트가 Suspense 컴포넌트

```
//fallback 속성에는 발생한 지연 시간 동안에 보여줄 컴포넌트를 지정할 수 있습니다.
//1. 특정 컴포넌트를 감싸줄 수 있습니다.
<React.Suspense fallback={<Loading />}>
    <TestComponent />
</React.Suspense>

//2. <Router /> 컴포넌트도 감싸줄 수 있습니다.
<React.Suspense fallback={<Loading />}>
  <Router>
     ……
   </Router>
</React.Suspense>
```

7.3 Lazy Loading 적용하기

❖의존 모듈 다운로드

```
//스피너 UI를 위한 react-spinner와 vite 프로젝트를 위한 의존모듈 설치
//p-min-delay 모듈은 의도적 지연 시간 발생을 위한 것이므로 실무에서는 사용하지 말 것
npm install react-spinners p-min-delay
npm install -D vite-plugin-webpackchunkname
```

❖vite.config.js 설정

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import { manualChunksPlugin } from 'vite-plugin-webpackchunkname'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react(), manualChunksPlugin()],
})
```

7.3 Lazy Loading 적용하기

❖ Loading 컴포넌트 확인

```
import React from 'react';
import { ScaleLoader } from 'react-spinners';

const Loading = () => {
    return (
        <div className="w-100 h-75 position-fixed">
            <div className="row w-100 h-100 justify-content-center align-items-center">
                <div className="col-6 text-center">
                    <h3>Loading</h3>
                    <ScaleLoader height="40px" width="6px" radius="2px" margin="2px" />
                </div>
            </div>
        </div>
    );
};

export default Loading;
```

7.3 Lazy Loading 적용하기

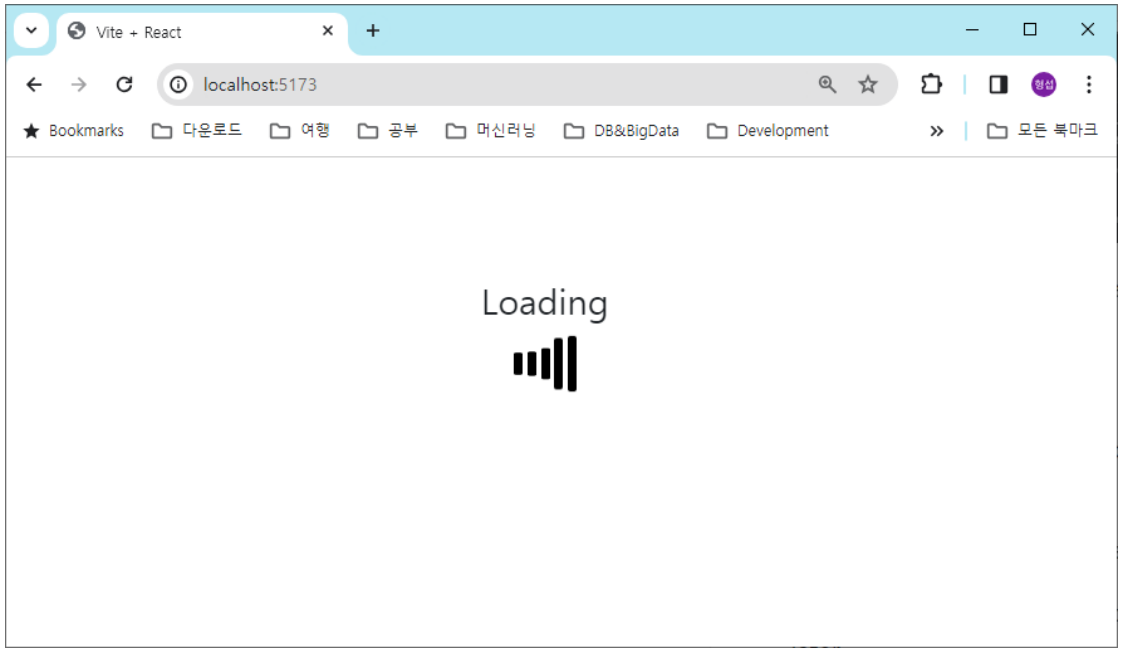
❖ App 컴포넌트 변경

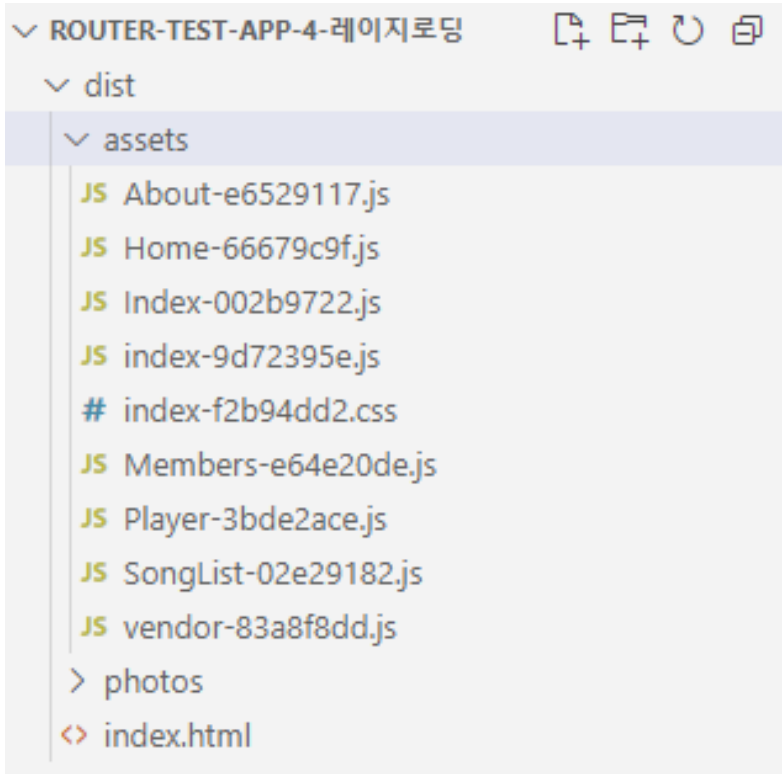
```
import Loading from './components/Loading';
import pMinDelay from "p-min-delay";

//const Home = React.lazy(() => import('./pages/Home'));
//의도적 지연시간 2초
const Home = React.lazy(() => pMinDelay(import('./pages/Home'), 2000));
const About = React.lazy(() => pMinDelay(import('./pages/About'), 2000));
const SongList = React.lazy(() => pMinDelay(import('./pages/SongList'),2000));
const Members = React.lazy(() => pMinDelay(import('./pages/Members'), 2000));
const Player = React.lazy(() => pMinDelay(import('./pages/Player'), 2000));
const SongIndex = React.lazy(() => pMinDelay(import('./pages/songs/Index'), 2000));

const App = () => {
    ......
    return (
      <React.Suspense fallback={<Loading />}>
          <Router>
              ......
          </Router>
      </React.Suspense>
    );
};
export default App;
```

7.3 Lazy Loading 적용하기

❖실행 결과

- 처음 컴포넌트의 청크를 로딩할 때 fallback UI 나타남
- 이후로는 fallback UI 나타나지 않음





- npm run build 하여 빌드된 청크파일 확인(오른쪽 그림 확인)

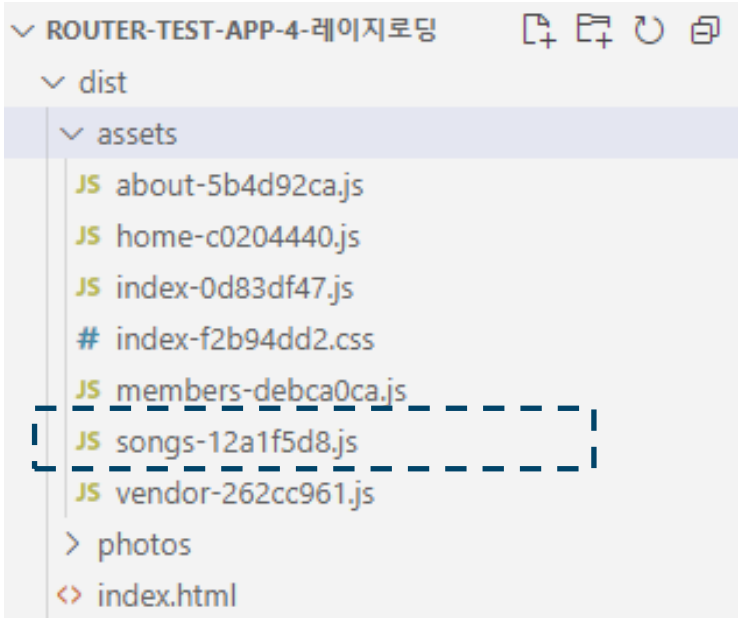
7.3 Lazy Loading 적용하기

❖청크파일 묶어주기

- 관련된 것, 동시에 로딩될 만한 것들은 동일한 청크이름으로 지정함

```
const Home = React.lazy(() => pMinDelay(import(/* webpackChunkName:"home" */ './pages/Home'), 2000));
const About = React.lazy(() => pMinDelay(import(/* webpackChunkName:"about" */ './pages/About'), 2000));
const SongList = React.lazy(() => pMinDelay(import(/* webpackChunkName:"songs" */ './pages/SongList'),2000));
const Members = React.lazy(() => pMinDelay(import(/* webpackChunkName:"members" */ './pages/Members'), 2000));
const Player = React.lazy(() => pMinDelay(import(/* webpackChunkName:"songs" */ './pages/Player'), 2000));
const SongIndex = React.lazy(() => pMinDelay(import(/* webpackChunkName:"songs" */ './pages/songs/Index'), 2000));
```

- npm run build 후 다시 확인



Q&A