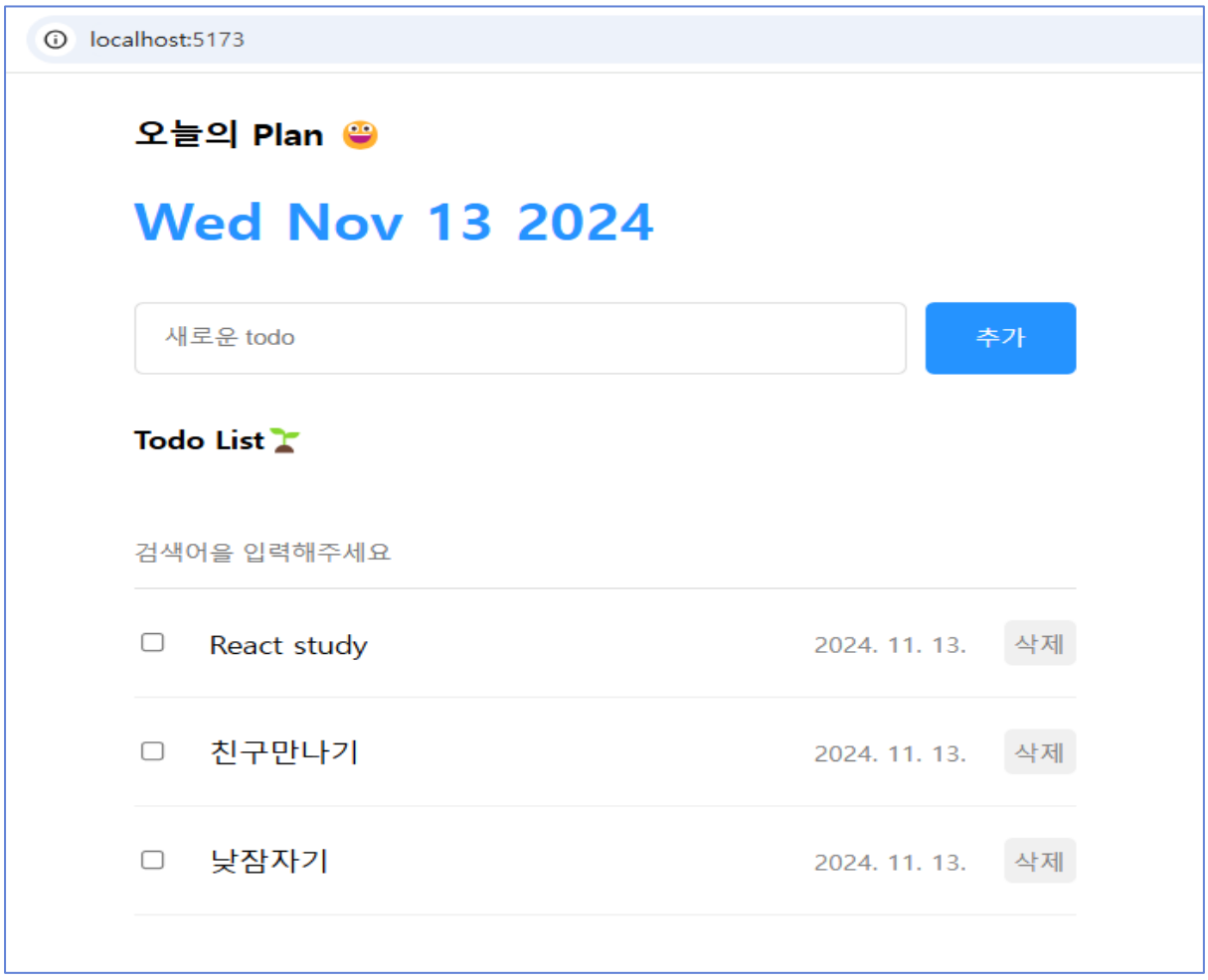
8253jang@daum.net
장희정

Todo-List Project

주요기능

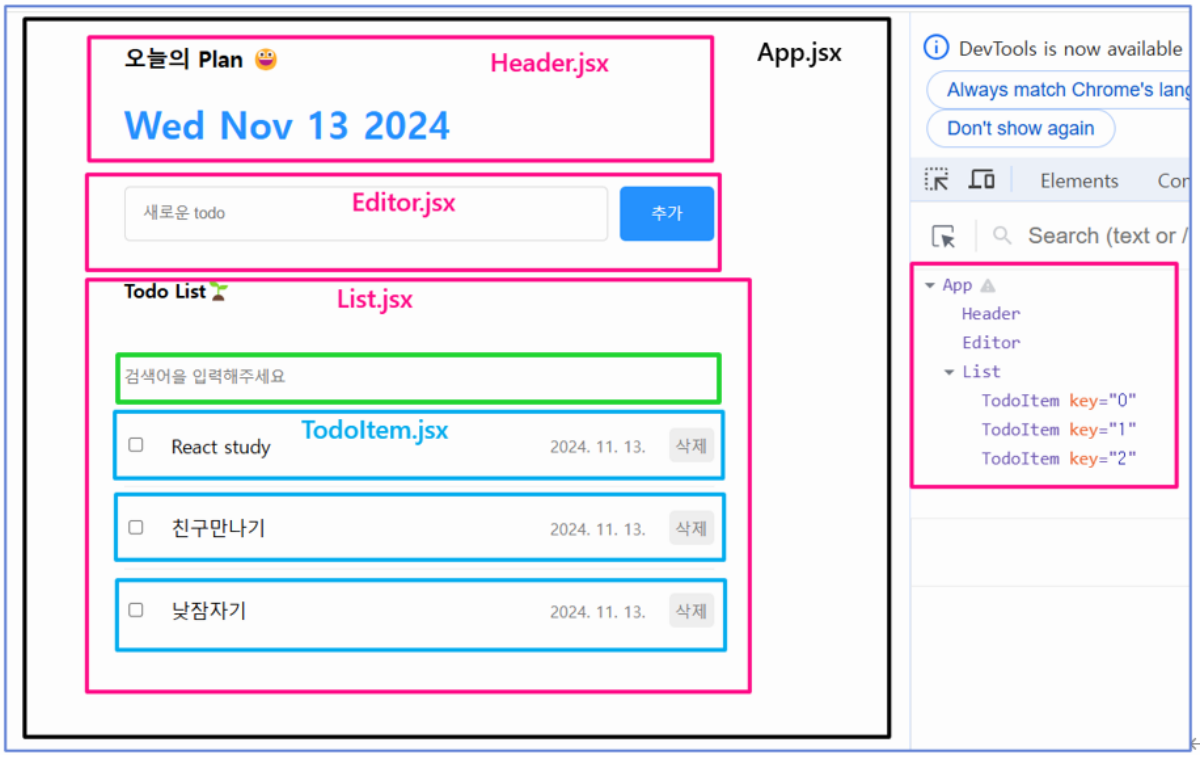
1)전체출력
2)추가(등록)
3)수정(checkbox 상태변경)
4)삭제
5)검색

Step01 - 전체 UI

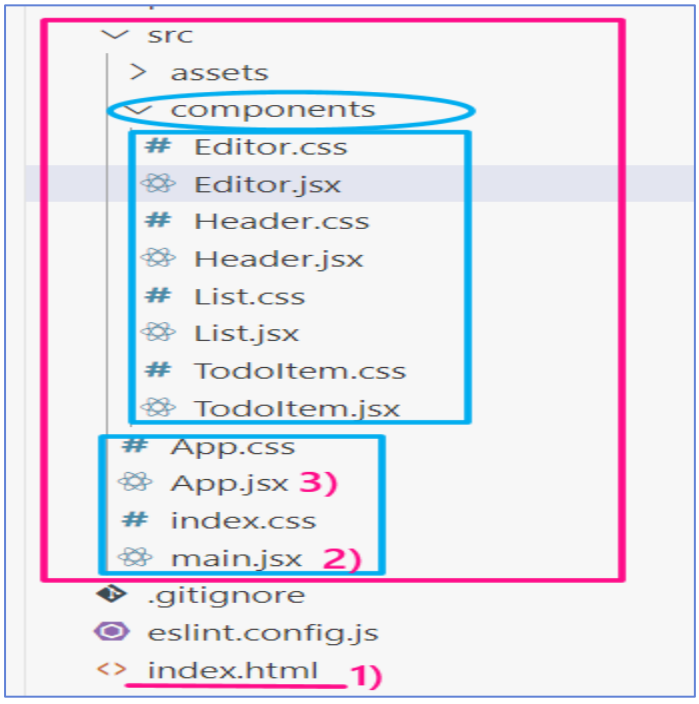


Component 구조

8253jang@daum.net
장희정

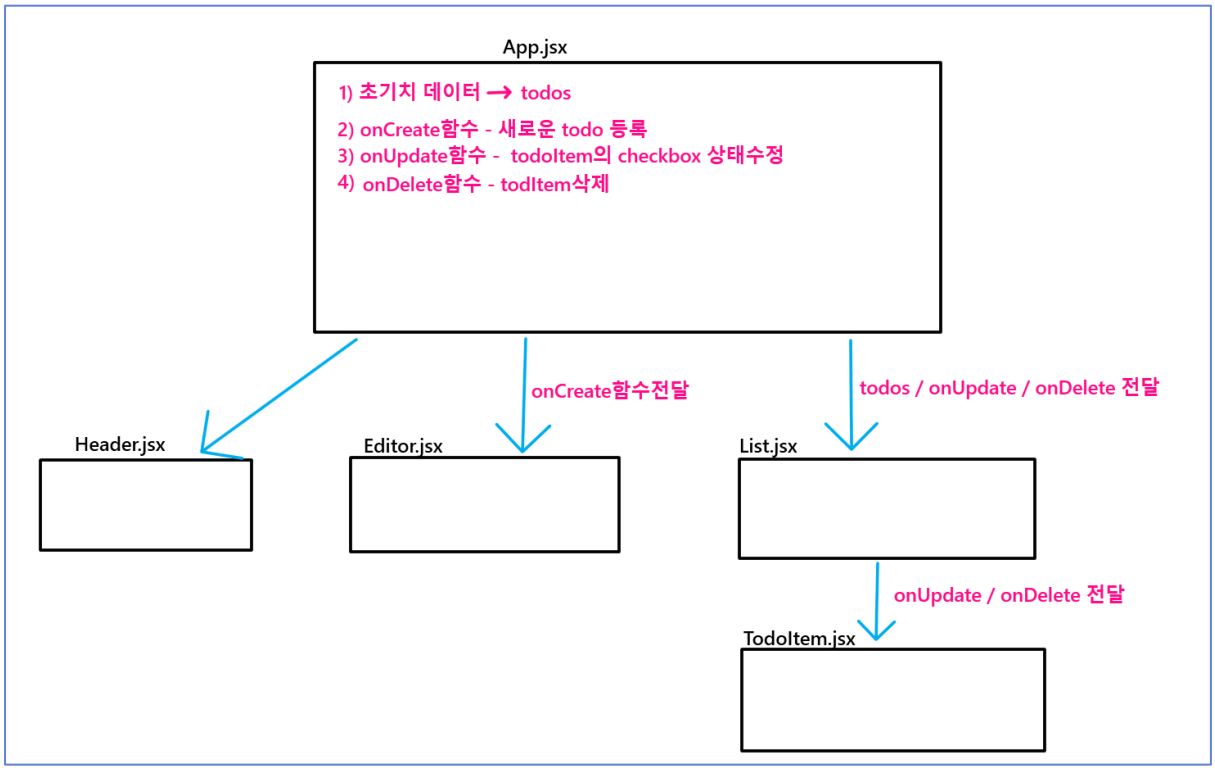


directory구조



8253jang@daum.net
장희정

props 전달과정



알아야 하는 개념

1) Component 분리
2) CSS적용
3) 이벤트처리
4) Props
5) useState
6) useRef
7) filter함수 , map함수

UI완성하기

Header.jsx

8253jang@daum.net
장희정

```
2  import "./Header.css";
3  const Header = () => {
4      return (
5          <div className="Header">
6              <h3>오늘의 Plan🌼</h3>
7              <h1>{new Date().toLocaleString()}</h1>
8          </div>
9      );
10 };
11
12 export default Header;
```

Window key + . 을 누르면 "이모지" 아이콘 나온다.

Header.css

```
1  .Header > h1{
2    color: rgb(37, 147, 255);
3  }
```

Editor.jsx

8253jang@daum.net
장희정

```jsx
import "./Editor.css";
import { useState } from 'react';
const Editor = () => {
    const [content, setCotent] = useState("");
    return (
        <div className="Editor">
            <input type="text" placeholder='새로운 todo' value={content}
            onChange={(e)=>setCotent(e.target.value)}/>
            <button>추가</button>
        </div>
    );
};

export default Editor;
```

Editor.css

```css
.Editor{
    display:flex;
    gap: 10px;
}

/*input 요소 css */
.Editor input{
    flex: 1; /*1은 부모요소에 벗어나지 않는 범위내에서 최대 크기로 나온다. */
    padding: 15px;
    border: 1px solid rgb(220,220,220);
    border-radius: 5px;
}

/*button요소 css*/
.Editor button{
    cursor: pointer;
    width: 80px;
    border: none;
    color: white;
    background-color: rgb(37,147,255);
    border-radius: 5px;
}
```

List.jsx

8253jang@daum.net
장희정

```jsx
import TodoItem from './TodoItem';
import "./List.css"
import { useState } from 'react';
const List = () => {
    const [search, setSearch] = useState("");
    return (
        <div className="List">
            <h4>Todo List 🌱</h4>
            <input placeholder='검색어를 입력해주세요.' value={search}
            onChange={(e)=>setSearch(e.target.value)} />

            <div className="todos_wrapper">
                <TodoItem />
                <TodoItem />
                <TodoItem />
            </div>

        </div>
    );
};

export default List;
```

List.css

```css
.List{
    display: flex;
    flex-direction: column;
    gap: 20px;
}

.List > input{
    width: 100%;
    border: none;
    border-bottom: 1px solid rgb(220, 220, 220);
    padding: 15px 0px;
}

.List > input:focus{
    outline: none;
    border-bottom: 1px solid rgb(37, 147, 255);
}

.List > .todos_wrapper{
    display: flex;
    flex-direction: column;
    gap: 20px;
}
```

8253jang@daum.net
장희정

TodoItem.jsx

```jsx
import "./TodoItem.css";
const TodoItem = () => {
    return (
        <div className="TodoItem">
            <input type="checkbox" />
            <div className="content">내용</div>
            <div className="date">날짜</div>
            <button>삭제</button>
        </div>
    );
};

export default TodoItem;
```

TodoItem.css

```css
.TodoItem{
    display: flex;
    align-items: center;/*열을 기준으로 가운데 정렬*/
    gap: 20px;
    padding-bottom: 20px;
    border-bottom: 1px solid rgb(240, 240, 240);
}

.TodoItem input{
    width: 20px;
}

.TodoItem .content{
    flex : 1;/*1은 부모요소에 벗어나지 않는 범위내에서 최대 크기로 나온다. */
 }

 .TodoItem .date{
    font-size: 14px;
    color: gray;
}

.TodoItem button{
   cursor: pointer;
   color:gray;
   font-size: 14px;
   border: none;
   border-radius: 5px;
   padding: 5px;
}

```

8253jang@daum.net
장희정

App.jsx

```jsx
import './App.css'
import Header from './components/Header'
import Editor from './components/Editor'
import List from './components/List'

function App() {
  return (
    <div className="App">
     <Header/>
     <Editor/>
     <List/>
    </div>
  )
}

export default App

```

App.css

```css
.App{
  display: flex;
  flex-direction: column;
  gap: 10px;
  width: 500px;
  margin: 0 auto;
}
```

UI 디자인 완성 후 모습

8253jang@daum.net
장희정

초기치 데이터 세팅하기

App.jsx

```
7  //랜더링이 될때 다시 실행되지 않아도 되기에 함수 밖에 선언한다.
8  const mockData = [
9    {id:0, isDone:false, content:"React study", date: new Date().getTime()},
10   {id:1, isDone:false, content:"친구만나기", date: new Date().getTime()},
11   {id:2, isDone:false, content:"낮잠자기", date: new Date().getTime()},
12 ];
13
14 function App() {
15   const [todos , setTodos] = useState(mockData);
16
17   return (
18     <div className="App">
19       <Header/>
20       <Editor/>
21       <List todos={todos}/>
22     </div>    List -> TodoItem 에서 사용 할 수 있게 props로 전달한다.
23   )
24 }
```

8253jang@daum.net
장희정

List.jsx

```jsx
const List = ({todos}) => {
    const [search, setSearch] = useState("");
    return (
        <div className="List">
            <h4>Todo List 🌱</h4>
            <input placeholder='검색어를 입력해주세요.' value={search}
            onChange={(e)=>setSearch(e.target.value)} />

            <div className="todos wrapper">
                {
                    todos.map((todo)=>{
                      return <TodoItem key={todo.id} {...todo} />
                    })
                }
            </div>
```

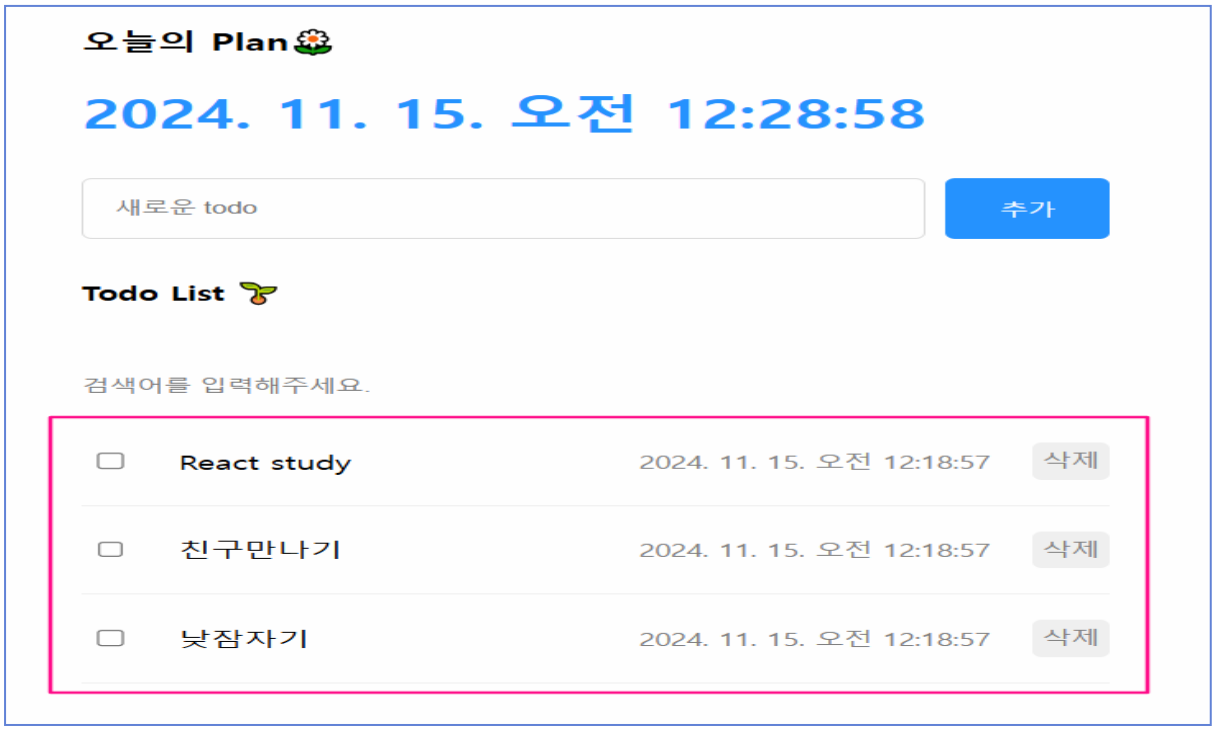
TodoItem.jsx

```jsx
const TodoItem = ({id, isDone, content , date}) => {
    const onChangeCheckbox=()=>{
       //수정하기(checkbox 상태변경)
    }

    return (
        <div className="TodoItem">
            <input type="checkbox" checked={isDone} onChange={onChangeCheckbox}/>
            <div className="content">{content}</div>
            <div className="date">{new Date(date).toLocaleString()}</div>
            <button>삭제</button>
        </div>
    );
};
```

적용 후 화면

8253jang@daum.net
장희정



Step02 - 추가하기 | 수정하기 | 삭제하기

기능에 해당하는 함수를 모든 Component에서 사용 할 수 있도록 부모 Component인 App.jsx에 선언한다.

```
18    //추가하기
19    const onCreate = ()=>{
20
21    }
22
23    //수정하기
24    const onUpdate = ()=>{
25
26    }
27
28    //삭제하기
29    const onDelete = ()=>{
30
31    }
```

8253jang@daum.net
장희정

선언된 함수를 자식 Component에 props로 전달한다.

App.jsx

```
return (
  <div className="App">
   <Header/>
   <Editor onCreate={onCreate}/>
   <List todos={todos}    onUpdate={onUpdate} onDelete={onDelete} />
  </div>
)
```

추가하기

```
function App() {
 const [todos , setTodos] = useState(mockData);
 const idRef = useRef(3); //재 랜더링이 되어도 내부적으로 값을 유지 할 수 있도록 Ref 사용
 //추가하기
 const onCreate = (content)=>{
   const newTodo={
     id:idRef.current++ , isDone:false, content:content, date: new Date().getTime()
   }

   setTodos([newTodo , ...todos ])
 }
```

위 코드에서 useRef()를 사용하는 이유는 새로운 할 일(todo)의 id를 내부적으로 유일한 값으로 관리하고, 이 값을 컴포넌트가 리렌더링될 때마다 유지하기 위함이다. 이 경우, 상태(state)보다 ref가 더 적합하다. **왜냐하면** ref는 컴포넌트 리렌더링과 관계없이 값이 유지되기 때문이다.

Editor.jsx

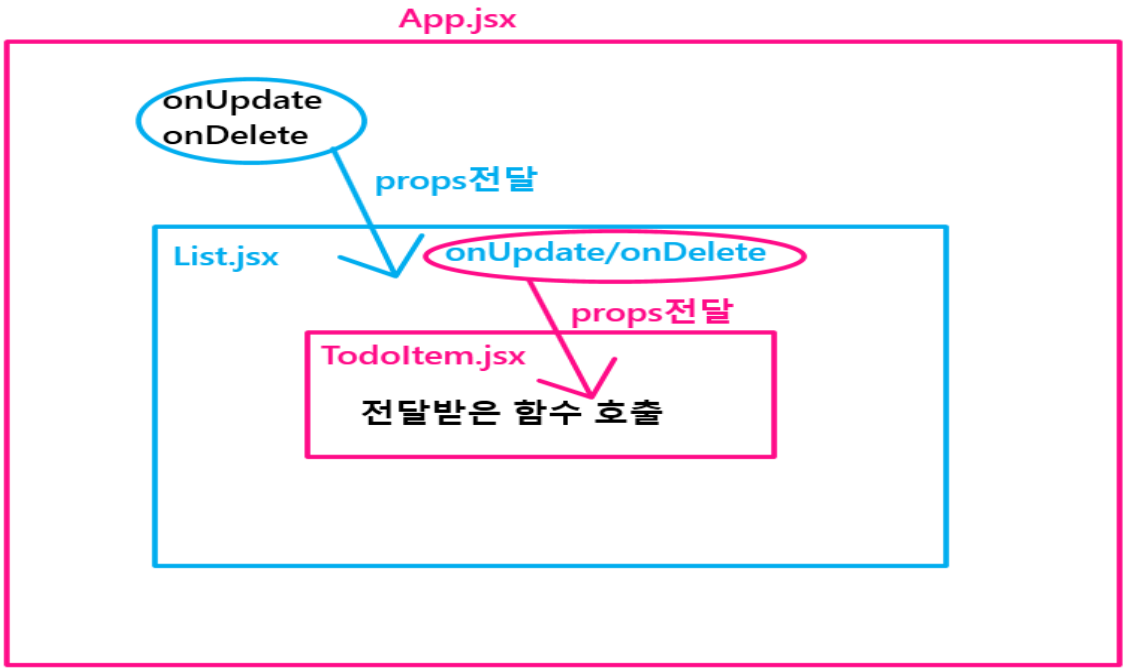
```
5  const Editor = ({onCreate}) => {
6      const [content, setContent] = useState("");
7      const contentRef = useRef(); //input Dom요소에 접근 하기 위한 Ref
8
9      //추가버튼 클릭
10     const onSubmit = ()=>{
11         if(content ===""){
12             contentRef.current.focus();
13             return;
14         }
15
16         //추가 기능 함수 호출(App.jsx파일에 있는 함수 호출된다.)
17         onCreate(content);
18         setContent("");
19     }
20
21     //엔터입력했을때 onsubmit 호출
22     const onkeydown = (e)=>{
23         if(e.keyCode===13){
24           onSubmit();
25         }
26      }
27
28     return (
29         <div className="Editor">
30             <input type="text" placeholder='새로운 todo' value={content}
31             onChange={(e)=>setContent(e.target.value)}  ref={contentRef}
32             onKeyDown={onkeydown}/>
33             <button onClick={onSubmit}>추가</button>
```

수정하기

Checkbox를 선택하면 선택된 id에 해당하는 todo의 상태(isDone)을 변경한다.

App.jsx파일에 선언되어 있는 onUpdate, onDelete 함수를 List에 전달하고 List 는 전달 받는 props를 다시 TodoItem에 props로 전달해야 한다.

8253jang@daum.net
장희정




```
<List todos={todos}   onUpdate={onUpdate} onDelete={onDelete} />
```

```
<TodoItem key={todo.id} {...todo}  onUpdate={onUpdate} onDelete={onDelete}/>
```

App.jsx파일 수정 함수

```
31  //수정하기
32  const onUpdate = (targetId)=>{ //TodoItem에서 호출할때 전달한 id
33    //todos state의 값들중에 targetId와 일치하는 todoitem의 isDone 변경
34    const updateTodos =  todos.map((todo)=>{
35      if(todo.id===targetId){
36        return {...todo , isDone : !todo.isDone}
37      }else{
38        return todo;
39      }
40    });
41
42    setTodos(updateTodos);
43
44    ////위 코드를 삼항연산자로 변경//////////////////////////
45    //setTodos(todos.map((todo)=>todo.id===targetId ? {...todo , isDone : !todo.isDone}:todo));
46  }
```

삭제하기

```
const TodoItem = ({id, isDone, content , date , onUpdate , onDelete }) => {
    const onChangeCheckbox=()=>{
       //수정하기(checkbox 상태변경)
       onUpdate(id);
    }

    //삭제 클릭했을때
    const onClickDeleteButton = ()=>{
        onDelete(id);
    }

    return (
        <div className="TodoItem">
            <input type="checkbox" checked={isDone} onChange={onChangeCheckbox}/>
            <div className="content">{content}</div>
            <div className="date">{new Date(date).toLocaleString()}</div>
            <button onClick={onClickDeleteButton}>삭제</button>
        </div>
    );
};
```

App.jsx onDelete 함수

```
//삭제하기
const onDelete = (targetId)=>{
  //인수 : todos배열에서 targetId와 일치하는 id를 갖는 요소만 삭제한 새로운 배열
  setTodos( todos.filter((todo)=>todo.id!==targetId) );
}
```

검색하기

text박스에 검색어를 입력하면 input에 state가 변경되면서 리랜더링이 된다. 리랜더링이 될 때 마다 filter함수를 이용해서 검색어에 해당하는 todo를 필터링 하여 화면에 랜더링한다.

```
7
8   //검색어를 입력했을때 검색어를 포함한 todo정보 조회
9   const getFilterData =()=>{
10     if(search===""){return todos;}
11
12    const searchedTodos =  todos.filter((todo)=>{
13       return todo.content.toLowerCase().includes(search.toLowerCase());
14     });
15
16    return searchedTodos;
17  }
18
19  //컴포넌트가 리랜더링 될때마다.. 호출되도록...
20  const filteredTods = getFilterData();
21
```

map 과 filter 의 차이

1)map 함수

map 함수는 배열의 각 요소에 대해 제공된 콜백 함수를 실행하고, 그 결과로 새로운 배열을 반환한다. 즉, 배열의 각 항목을 변환하는 데 사용된다.

목적 : 배열의 각 요소를 변환하여 새로운 배열을 생성.
반환값 : 새로운 배열(변환된 값으로 구성).
원본 배열 : 변하지 않음.

2) filter 함수

filter 함수는 배열의 **각 요소에 대해 제공된 콜백 함수가 true를 반환하는 경우에만** 그 요소를 새로운 배열에 포함시킨다. 즉, 배열에서 조건에 맞는 요소만 추출하는 데 사용된다.

목적 : 배열에서 조건을 만족하는 요소만 필터링하여 새로운 배열을 생성.
반환값 : 조건을 만족하는 요소들로 이루어진 새로운 배열.
원본 배열 : 변하지 않음.

8253jang@daum.net
장희정

Step03 – TodoList 업그레이드 하기

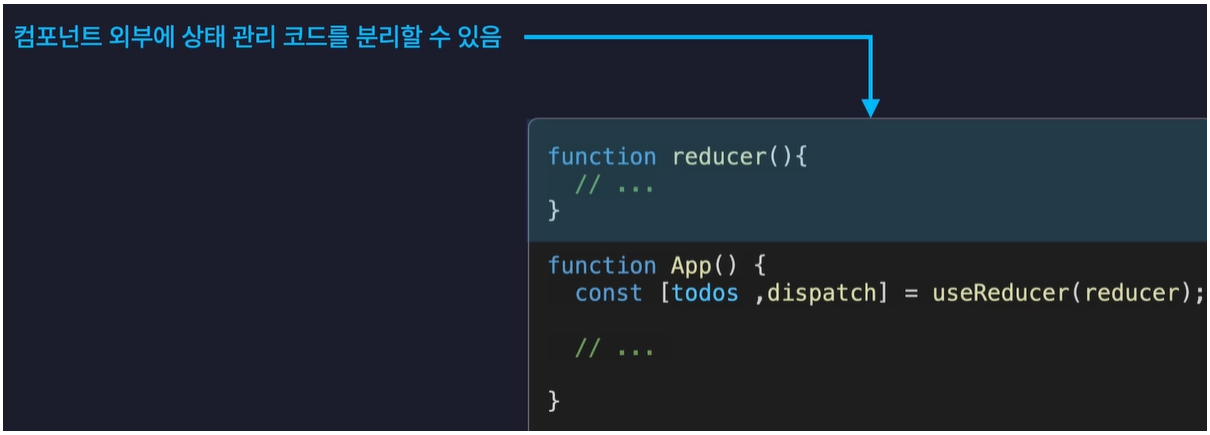
1) useReducer() - useState를 이용한 복잡한 상태 로직을 useReducer로 변경
2) useMemo() – 연산 최적화
3) React.memo() – 컴포넌트 랜더링 최적화
4) useCallback() – 함수 재 생성 방지
5) useContext() – 전역적으로 상태를 공유 하고 관리

1)todoList 프로젝트에 useReducer 적용하기

React의 useReducer()는 React의 상태 관리 훅 중 하나로, **복잡한 상태 로직**을 처리하거나 여러 값이 연관된 상태를 다룰 때 유용하다 . useState는 상태를 간단하게 관리하는 데 적합하지만, 상태가 복잡하거나 여러 개의 상태가 서로 의존하는 경우 useReducer가 더 적합할 수 있다.

컴포넌트 내부에 새로운 State를 생성하는 Reat Hook으로 **모든 useState는 useReducer로 대체가 가능한다.**

useReducer는 상태관리 코드를 컴포넌트 외부로 분리 할 수 있다



우리가 만든 todoList 프로젝트의 useState는 상태 관리하는 코드들이 App 컴포넌트 내부에 존재 해야 하기 때문에 소스가 복잡하다. 사실 App.jsx 는 UI를 랜더링 하기 위한 컴포넌트 인데 컴포넌트안에 복잡한 함수가 많이 들어 있다. 이것을 useReducer를 이용해서 외부로 분리해보자.

사용방법

```
import  { useReducer } from 'react';

const [ state , dispatch]  = useReducer( reducer , initialState )
```

useReducer 함수 2 개의 인자

리듀서 함수 (reducer) : 상태를 어떻게 업데이트할지를 정의하는 함수

초기 상태 (initialState) : 상태의 초기값

useReducer 는 2 가지 값을 반환

현재 상태(state) : 상태의 현재 값

디스패치 함수 (dispatch): 액션을 전달하여 상태를 업데이트하는 함수

useReducer의 장점

복잡한 상태 관리 : 상태가 복잡하거나 여러 개의 값이 연관된 경우 useReducer 는 상태 업데이트를 더 예측 가능하고 깔끔하게 관리할 수 있게 도와준다.

다양한 액션 처리 : 리듀서 함수에서 다양한 액션 타입을 처리할 수 있어, 복잡한 상태 변경 로직을 하나의 함수로 통합하여 관리할 수 있다.

읽기 쉬운 코드 : 상태를 직접적으로 수정하는 것보다, 상태 업데이트가 항상 명확한 액션을 통해 이루어지기 때문에 코드가 더 읽기 쉽고 유지보수가 용이하다.

디버깅에 유리 : 상태 업데이트 로직이 명확한 액션을 통해 이루어지기 때문에, 디버깅이 용이하고 DevTools 같은 툴을 이용한 상태 추적도 가능하다.

예시) Exam.jsx

```jsx
import { useReducer } from 'react';

const reducer =(state, action)=>{//현재상태값, action
    console.log(state, action);

    switch(action.type){
      case "UP" : return state + action.data;
      case "DOWN" : return state - action.data;
      default: return state;
    }
}

const Exam = () => {
    const [state, dispatch ] = useReducer(reducer , 0);

    const onClickPlus =()=>{
        dispatch({
            type:"UP",
            data:1
        });
    }

    const onClickMinus =()=>{
        dispatch({
            type:"DOWN",
            data:1
        });
    }
    return (
        <div>
            <h1>{state}</h1>
            <button onClick={onClickPlus}>+</button>
            <button onClick={onClickMinus}>-</button>
        </div>
    );
};
```

*reducer는 상태를 어떻게 업데이트 할지를 정의하는 함수

*dispatch는 액션을 전달하여 상태를 업데이트하는 함수

버튼이 클릭되면 dispatche는 액션 객체를 받아 reducer를 호출하게된다.

App.jsx 를 변경해보자.

```jsx
const reducer =(state, action)=>{
    switch(action.type){
      case "CREATE" : return [action.data, ...state];
      case "UPDATE" : return state.map((todo)=>todo.id===action.targetId ? {...todo , isDone : !todo.isDone} : todo);
      case "DELETE" : return state.filter((todo)=>todo.id!==action.targetId);
      default : return state;
    }
}
```

```
26  function App() {
27    //const [todos , setTodos] = useState(mockData);
28    const [ todos , dispatch]=useReducer(reducer, mockData)
29
30    //id의 값은 화면을 갱신 하려는 용도가 아니고 내부적으로 값을 유지하기 용도이므로  ref사용한다.
31    const idRef = useRef(3); //재 랜더링이 되어도 내부적으로 값을 유지 할 수 있도록 Ref 사용
32
33    //추가하기
34    const onCreate = (content)=>{
35      dispatch({
36       type:"CREATE",
37       data:{
38         id: idRef.current++,
39         isDone:false,
40         content:content,
41         date: new Date().getTime(),
42       }
43      })
44    }
45
46    //수정하기
47    const onUpdate = (targetId)=>{ //TodoItem에서 호출할때 전달한 id
48      dispatch({
49        type:"UPDATE",
50        targetId
51       })
52    }
53
54    //삭제하기
55    const onDelete = (targetId)=>{
56      dispatch({
57        type:"DELETE",
58        targetId:targetId,
59       });
60    }
61
```

최적화(Optimization)

웹 서비스의 성능을 개선하는 모든 행위를 말한다.

1)일반적인 웹 서비스 최적화

-이미지, 폰트, 코드 파일 등의 정적 파일 로딩 개선
-불필요한 네트워크 요청 줄임

2) React App 내부의 최적화

-컴포넌트 내부의 불필요한 연산 방지

8253jang@daum.net
장희정

-컴포넌트 내부의 불필요한 함수 재 생성 방지

-컴포넌트의 불필요한 리렌더링 방지

todoList 프로젝트에 아래와 기능을 추가 해보자.

1)전체 todos의 개수
2)todos중에 isDone이 true인 todo의 개수
3)todos중에 isDone이 false인 todo의 개수

-List.jsx파일 수정

```
22  const getAnalyzedData =()=>{
23      console.log("getAnalyzedData call");
24
25      const totalCount = todos.length;
26      const doneCount = todos.filter((todo)=>todo.isDone).length;
27      const notDoneCount = totalCount - doneCount;
28
29     // return {totalCount:totalCount,doneCount:doneCount, notDoneCount:notDoneCount}
30     return {totalCount,doneCount, notDoneCount}
31  }
32
33  const {totalCount, doneCount ,notDoneCount } = getAnalyzedData();
34
35  ////////////////////////////////////////
36
37   return (
38       <div className="List">
39           <h4>Todo List 🌱</h4>
40           <div>
41               <div>total : {totalCount}</div>
42               <div>doneCount : {doneCount}</div>
43               <div>notDoneCount : {notDoneCount}</div>
44           </div>
```

위 코드의 문제점은?

연산을 하는 getAnalyzedData() 함수가 불 필요하게 리렌더링이 될 때마다 같은 작업을 반복한다. Todos에 대한 변화(추가, 삭제, 수정)에 대해서만 함수가 호출되고 그 이외에 검색을 할 때는 todos에 대한 연산결과 변화가 없기 때문에 작업을 또 할 필요가 없다. **이런 경우 useMemo() 를 이용하여 메이제이션을 하여 불필요한 연산을 최적화 할 수 있다.**

해결책은?

2) useMemo를 이용한 연산 최적화

8253jang@daum.net
장희정

"메모이제이션" 기법을 기반으로 불필요한 연산을 최적화 하는 React Hook

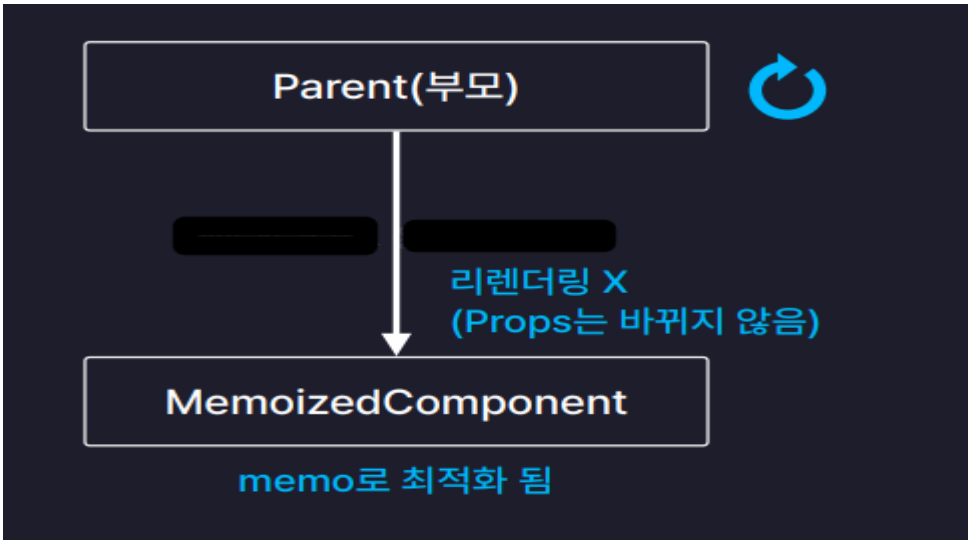
List.jsx파일 수정

```
38
39  const {totalCount, doneCount ,notDoneCount } = useMemo(()=>{
40          console.log("getAnalyzedData call");
41
42          const totalCount = todos.length;
43          const doneCount = todos.filter((todo)=>todo.isDone).length;
44          const notDoneCount = totalCount - doneCount;
45
46         return {totalCount,doneCount, notDoneCount}
47
48  } , [todos]);
49
50
51      return (
52          <div className="List">
53              <h4>Todo List 🌱</h4>
54              <div>
55                  <div>total : {totalCount}</div>
56                  <div>doneCount : {doneCount}</div>
57                  <div>notDoneCount : {notDoneCount}</div>
58              </div>
```

3) React.memo()를 이용한 최적화

React.memo()는 React 에서 고차 컴포넌트(Higher-Order Component, HOC)로, 컴포넌트의 렌더링 성능을 최적화하는 데 사용된다. **주로 컴포넌트가** 같은 props **로 반복 렌더링될 때 불필요한 렌더링을 방지하기 위해 사용된다**

React.memo()는 컴포넌트를 감싸는 함수로, 이 함수는 컴포넌트가 렌더링될 때 전달되는 props 가 이전 props 와 **다를 경우에만** 컴포넌트를 재렌더링한다. 즉, props 가 변하지 않으면 이전에 렌더링한 결과를 재사용하여 렌더링 성능을 최적화한다.



todoList 프로젝트에 React.memo() 적용해보자.

Header.jsx 컴포넌트는 App.jsx 컴포넌트가 리 랜더링 될때마다 다시 호출 될 필요가 없다.
React.memo를 이용해서 최적화를 시켜보자

```jsx
import React, { memo } from 'react';
import "./Header.css";
const Header = () => {
    return (
        <div className="Header">
            <h3>오늘의 Plan</h3>
            <h1>{new Date().toLocaleString()}</h1>
        </div>
    );
};

/*const memoizedHeader = memo(Header)
export default memoizedHeader;*/

export default memo(Header);
```

코드를 완성 한 후 실행 해보면 Header.jsx 컴포넌트가 불필요하게 리랜더링 되지 않는 것을 확인 할 수 있다.

두번째 ,

Todos의 TodoItem이 변경 될 때 마다 모든 TodoItem 컴포넌트가 호출된다.

TodoItem에도 React.memo() 를 적용해보자.

```jsx
export default memo(TodoItem);
```

하지만,

적용이 안 된다. React.memo()는 props를 얕은 비교를 하기 때문에서 함수, 배열, 객체가 props로 전달되면 항상 새로운 주소로 전달되어 최적화가 되지 않는다. Props가 불변 객체여야 최적화 효과가 제대로 나타난다. 아니면, 아래와 같이 직접 비교하는 커스텀 마이징이 필요하다.

```jsx
export default memo(TodoItem, (prevProps , nextProps)=>{
    //리턴값에 Props가 바뀌었는지 안 바뀌었는지 판단한다.
    // true -> Props 바뀌지 않음 (리랜더링 안됨)
    // false -> Props 바뀜 (리랜더링 됨)

    if(prevProps.id !== nextProps.id) return false;
    if(prevProps.isDone !== nextProps.isDone) return false;
    if(prevProps.content !== nextProps.content) return false;
    if(prevProps.date !== nextProps.date) return false;

    return true; //리랜더링 안됨
});
```

그러나
위 처럼 직접 커스텀 마이징을 하면 속성이 변경되거나 추가되면 매번 소스를 수정해야하기에 유지보수에 어려움이 생긴다.

그래서,

4)useCallback()를 이용하여 개선 해 보자.

useCallback()은 React 훅(Hook) 중 하나로, 주로 **함수 컴포넌트 내에서 함수의 참조를 메모이제이션**(memoization)하는 데 사용된다. 즉, 동일한 **의존성 배열**이 주어졌을 때, **불필요한 함수 재생성**을 방지하고 기존의 함수 참조를 재사용할 수 있게 도와준다. 이를 통해 성능 최적화를 유도할 수 있다.
useCallback()은 주로 자식 컴포넌트에 함수를 props로 전달할 때, 불필요하게 매번 새로운 함수가 생성되는 문제를 해결하기 위해 사용된다.

사용법

```
import { useCallback } from 'react';

const handleFun = useCallback ( 콜백함수 , [의존성배열] )
```

useCallback()의 2 개의 인자

콜백 함수 : 메모이제이션 할 함수
의존성 배열 : 이 배열 안의 값들이 변경될 때만 콜백 함수가 새로 생성

App.jsx수정

```
38  //onCreate, onUpdate, onDelete 함수를 useCallback을 이용해서 만들어보자
39  //함수를 자식 컴포넌트에 props로 전달할때마다 자식이 리랜더링 되는것을 막 을수 있다.
40  const onCreate = useCallback((content)=>{
41    console.log("onCreate content = " + content)
42      dispatch({
43        type:"CREATE",
44        data:{
45          id: idRef.current++,
46          isDone:false,
47          content:content,
48          date: new Date().getTime(),
49        }
50      });
51    } , []);
52
53  const onUpdate = useCallback((targetId)=>{
54    console.log("onUpdate targetId = " + targetId)
55    dispatch({
56      type:"UPDATE",
57      targetId
58     })
59
60  } , []);
61
62
63  const onDelete = useCallback((targetId)=>{
64    console.log("onDelete targetId = " + targetId)
65    dispatch({
66      type:"DELETE",
67      targetId:targetId,
68     });
69  } , []);
```

최적화는

최적화가 꼭 필요할 것 같은 연산이나 함수들에게만 선택적으로 적용하는 것이 좋다.

최적화에 관련된 함수 또한 최적화를 위한 연산이나 기억 해 두어야 하는 작업들이 필요하다. 단순한 작업을 위해 최적화 함수를 사용하면 오히려 더 많은 연산과 메모리를 차지 할 수 있다.

최적화는 기능을 완성 한 후에 최적화를 적용하는 것이 좋다.

참고(**When to use useMemo, useCallback)**

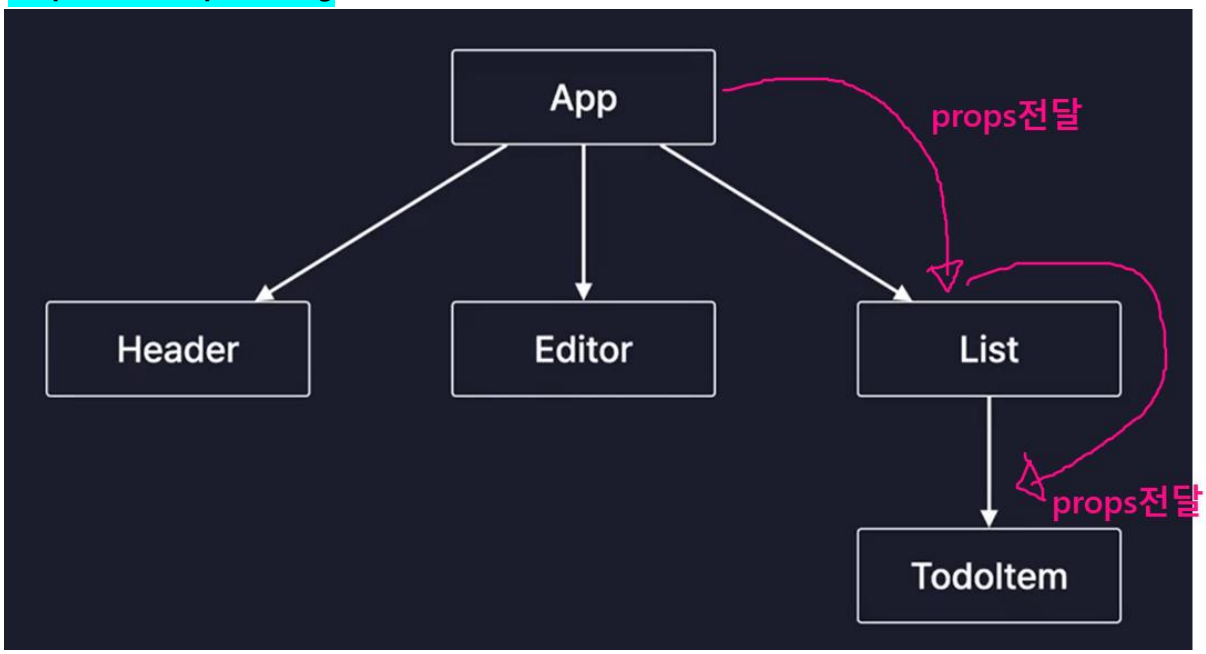
https://goongoguma.github.io/2021/04/26/When-to-useMemo-and-useCallback/

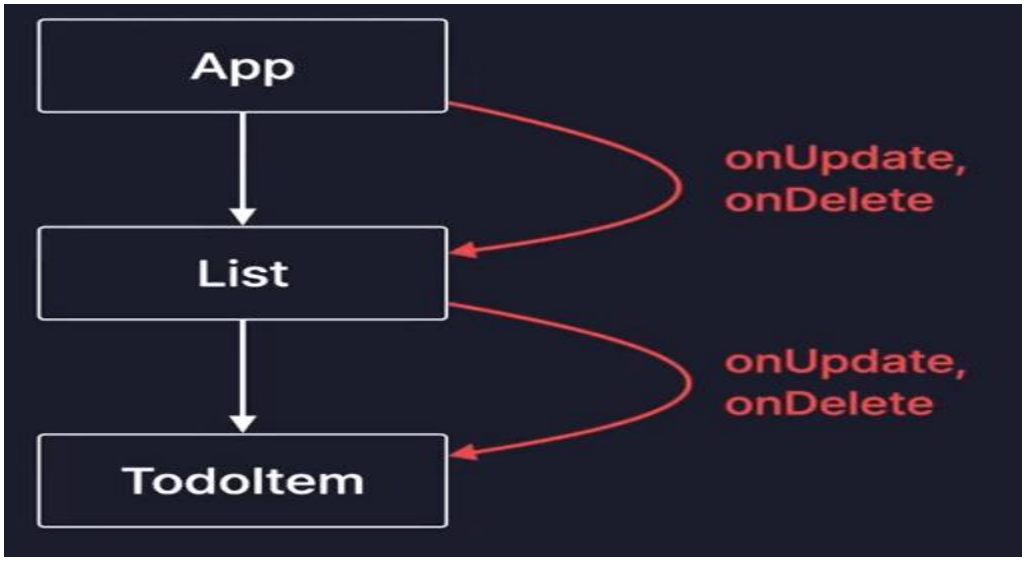
8253jang@daum.net
장희정

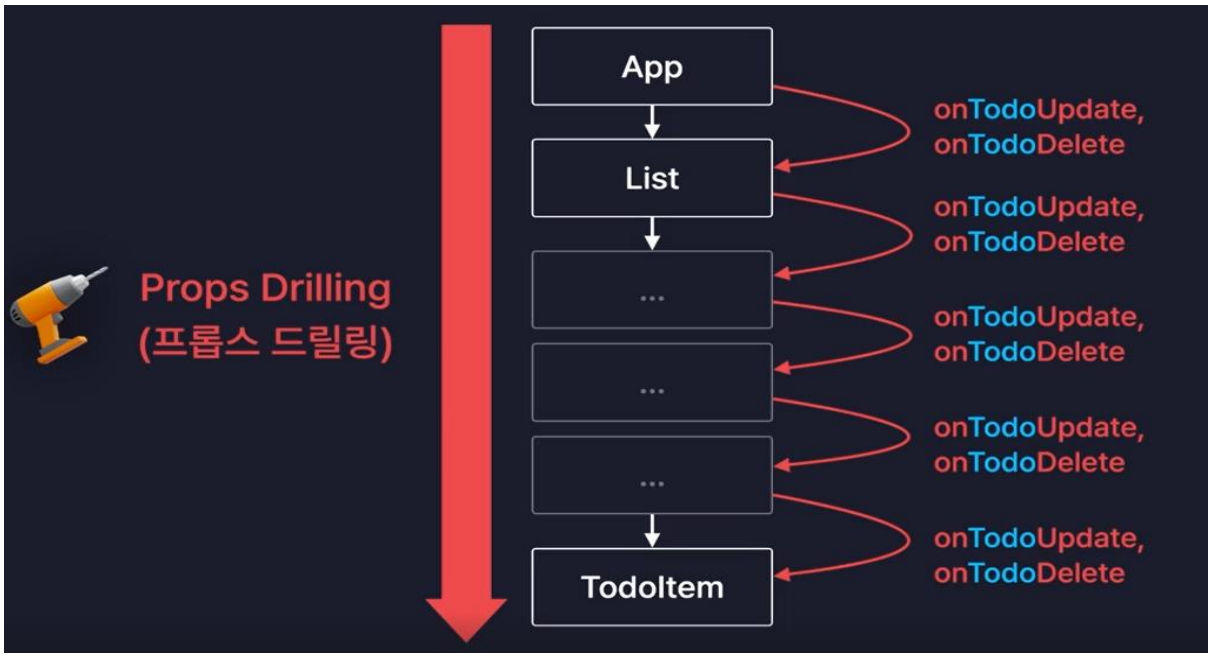
5)React Context

컴포넌트 간의 데이터를 전달하는 또 다른 방법으로 **기존의 Props가 가지고 있던 단점을 해 결 할 수 있다.**

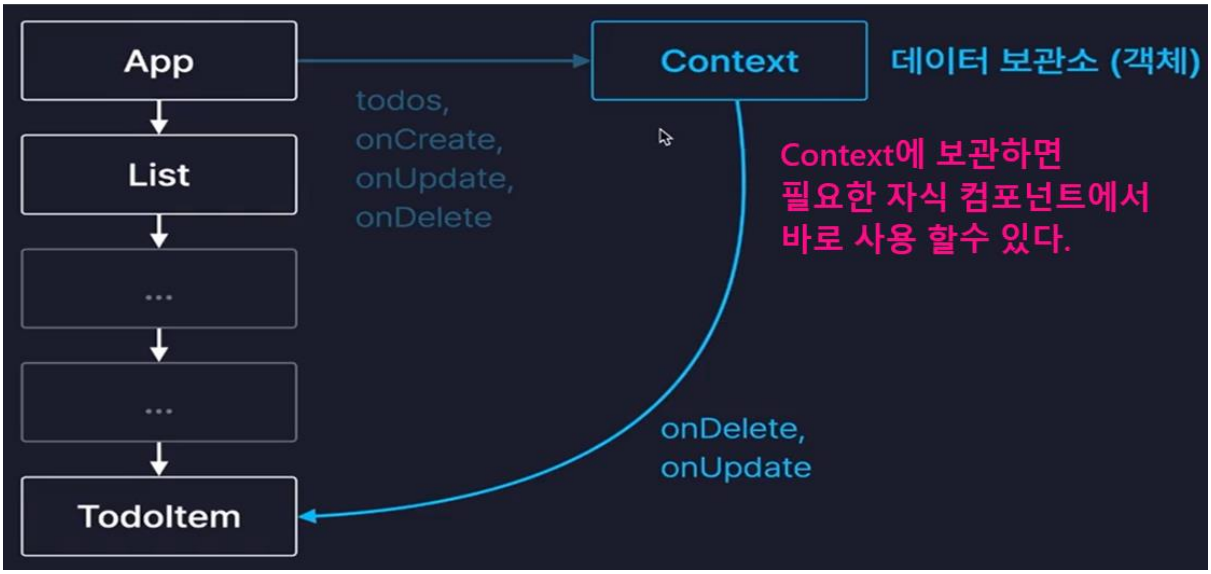
Props단점 : Props Drilling



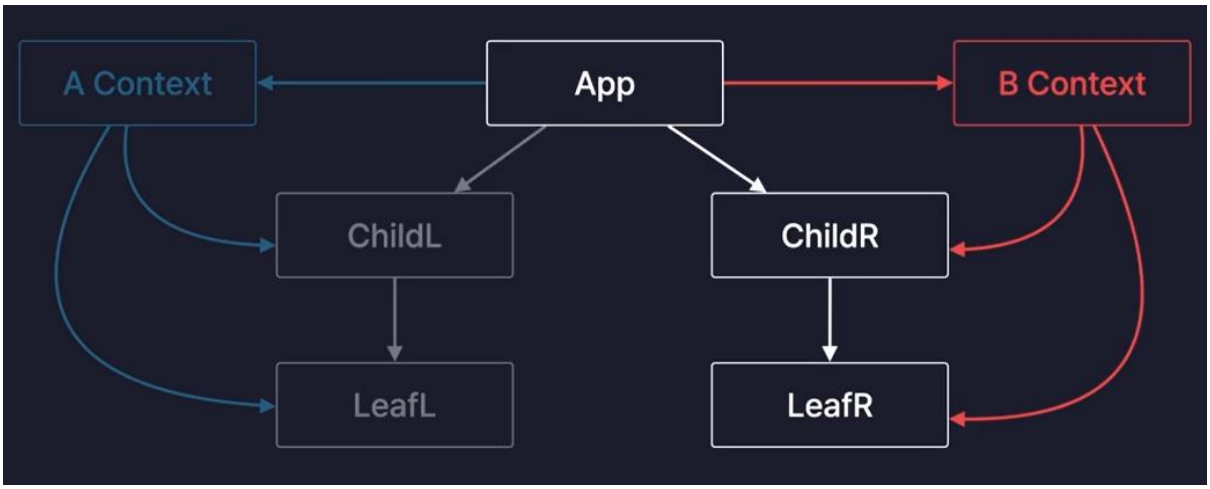




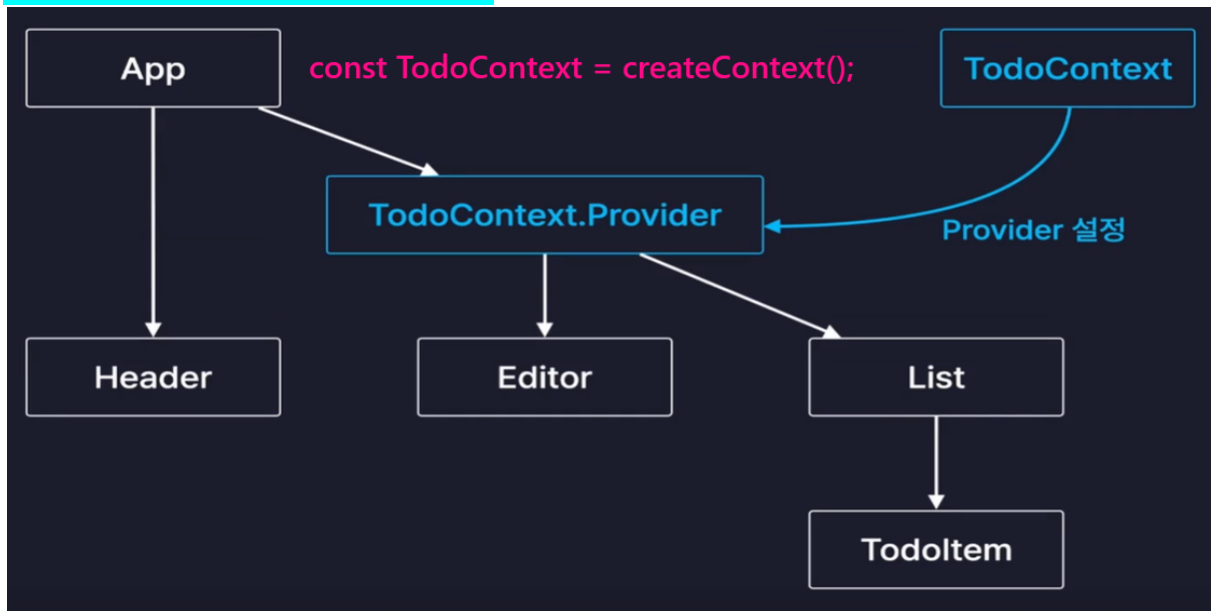
React의 Context를 이용하면



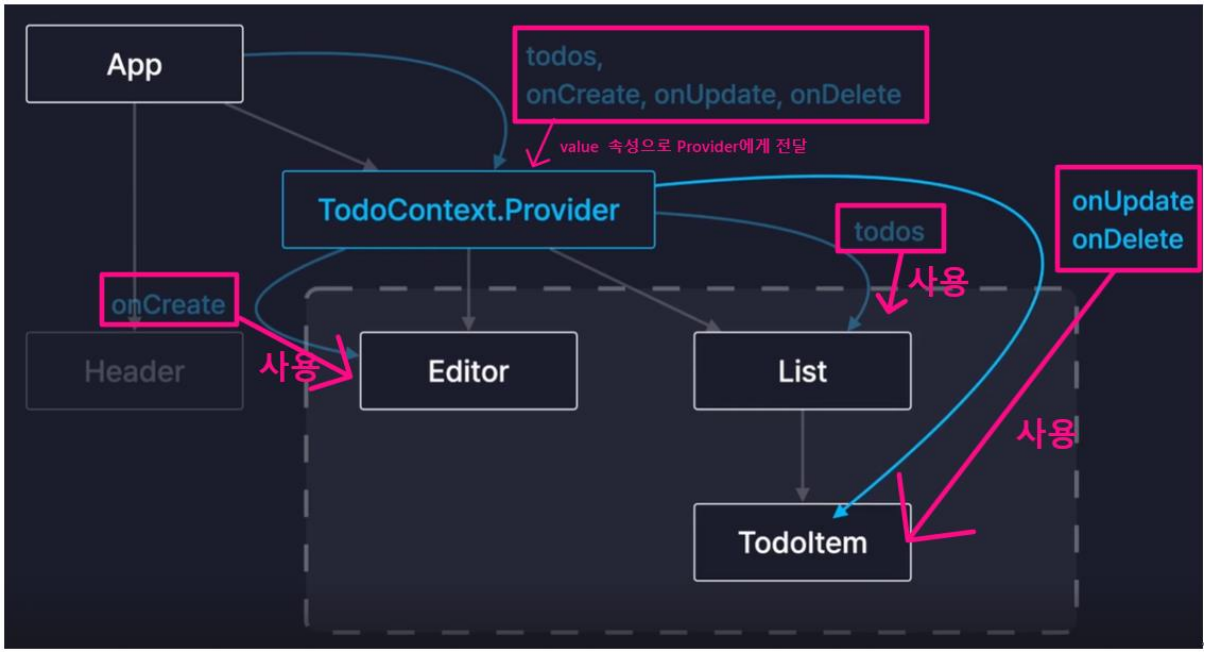
Context는 필요하면 여러 개를 만들어서 사용 할 수 있다.



우리 프로젝트에 Context를 적용 해보자.



8253jang@daum.net
장희정



App.jsx문서 – Context 생성하기

```
31  // context 사용하기
32  export const TodoContext = createContext();
```

```
77  return (
78    <div className="App">
79      <Header/>
80      <TodoContext.Provider value={{todos, onCreate, onUpdate, onDelete}}>
81        <Editor />
82        <List />
83      </TodoContext.Provider>
84    </div>
85  )
86  }
```

자식 component에서 useContext로 사용하기

-Editor.jsx 는 onCreate 필요

```
const Editor = () => {
    /*const data = useContext(TodoContext);
    console.log(data);*/

    const {onCreate} = useContext(TodoContext);
```

-List.jsx 컴포넌트는 todos 필요

```
8  const List = () => {
9      const {todos} = useContext(TodoContext);
10
```

```
<div className="todos_wrapper">
    {
        //todos.map((todo)=>{
        filteredTods.map((todo)=>{
          return <TodoItem key={todo.id} {...todo}  />
        })
    }
</div>
```

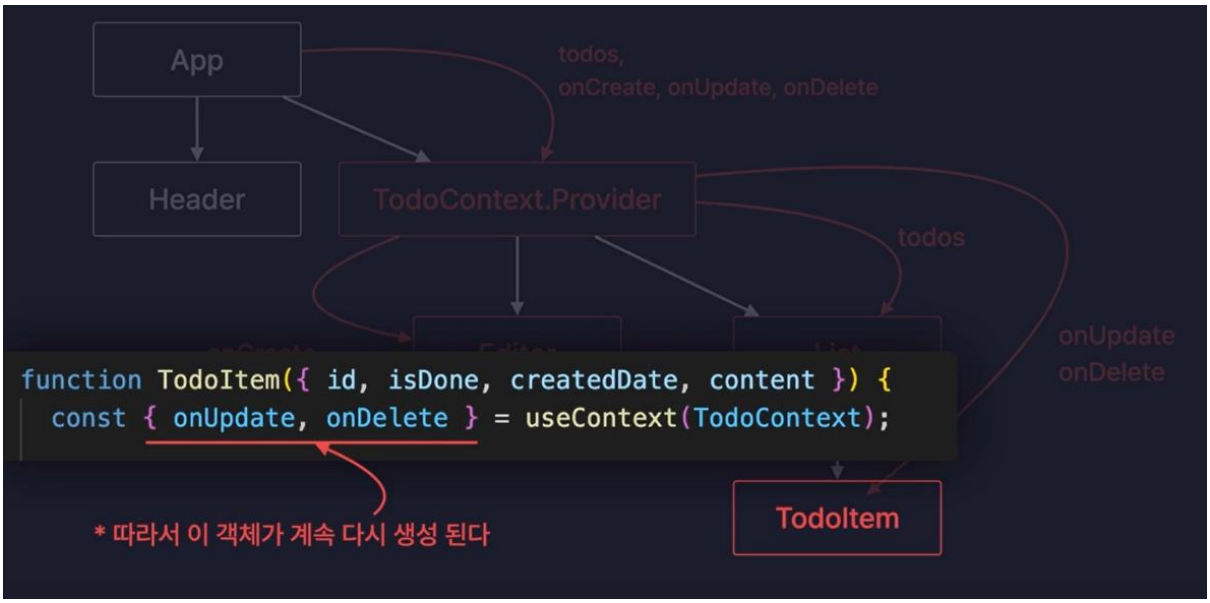
-TodoItem.jsx 컴포넌트는 onUpdate, onDelete 필요

```
6  const TodoItem = ({id, isDone, content , date }) => {
7
8      const {onUpdate, onDelete } = useContext(TodoContext);
9
10     console.log(id+"-> 랜더링...")
11
```

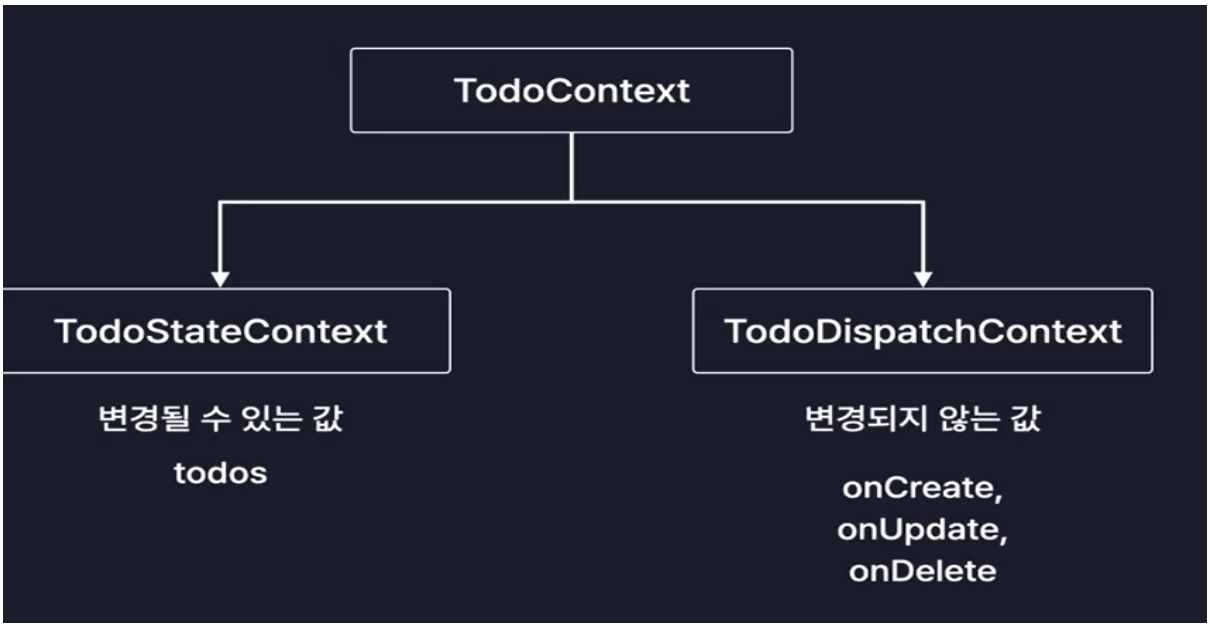
코드를 완성 한 후 실행 해보면 모든 기능은 잘 동작 되지만 하나의 특정 아이템의 체크박스를 on or off 를 해보면 이전에 useCallback과 memo로 적용 해 놓았던 TodoItem에 적용한 최적화가 되지 않는 것을 확인 할 수 있다.

이유는 TodoContext.Provider도 App컴포넌트로부터 전달받는 Props 이기 때문에 전달되는 props가

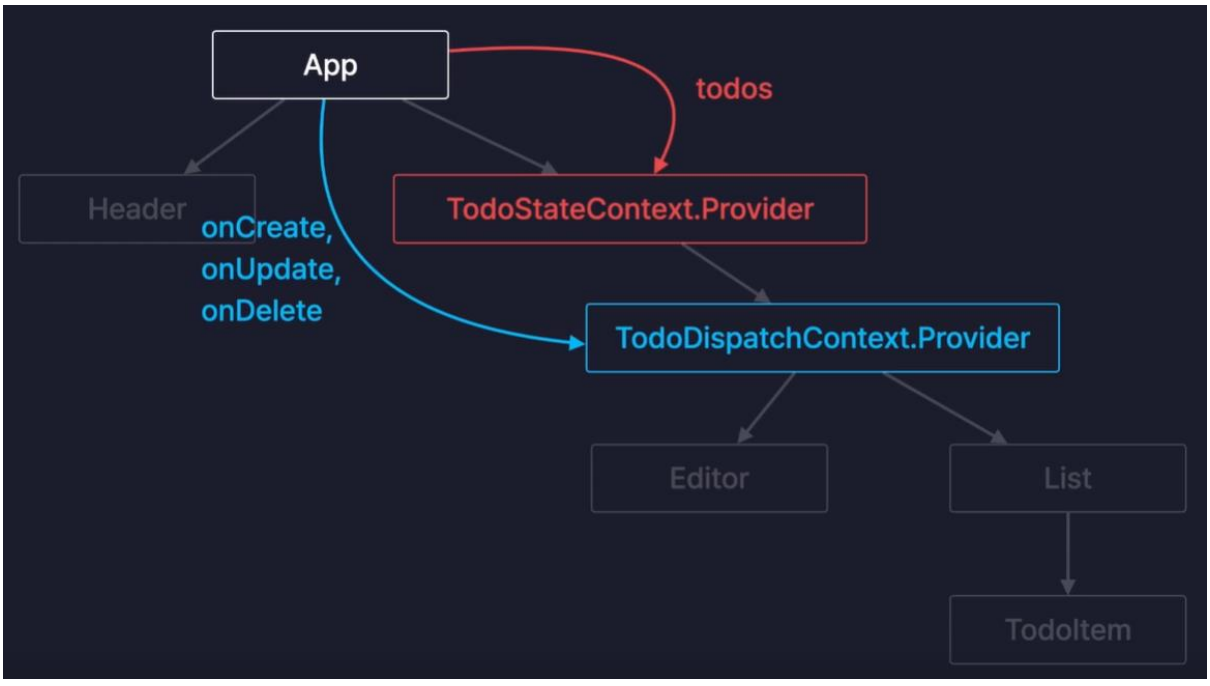
바뀌게 되면 다시 리랜더링이 된다. Todos에 추가, 삭제, 수정이 되면 todos가 변경이 되고 변경이 되면 Provider에게 전달되는 객체 자체가 다시 생성 되기 때문이다.



해결 방법 – 두개의 Context로 분리 한다.



Context 분리하기



App.jsx 수정해보자

```
31  // context 사용하기
32  export const TodoStateContext = createContext();
33  export const TodoDispatchContext = createContext();
```

```
76    todos state가 변경되면 onCreate, onUpdate,onDelete가 다시 생성되게 된다. 그래서 todos state가 변경되어도 다시 생성하지 않도록 useMemo 를 사용한다.
77    const memoizedDispatch = useMemo(()=>{ return {onCreate, onUpdate, onDelete} } , []);
78
79    return (
80      <div className="App">
81       <Header/>
82       <TodoStateContext.Provider value={todos}>
83        <TodoDispatchContext.Provider value={memoizedDispatch}>
84          <Editor />
85          <List />
86        </TodoDispatchContext.Provider>
87       </TodoStateContext.Provider>
88      </div>
89    )
90  }
```

Editor.jsx수정

```
 9  const Editor = () => {
10
11      const {onCreate} = useContext(TodoDispatchContext);
12
```

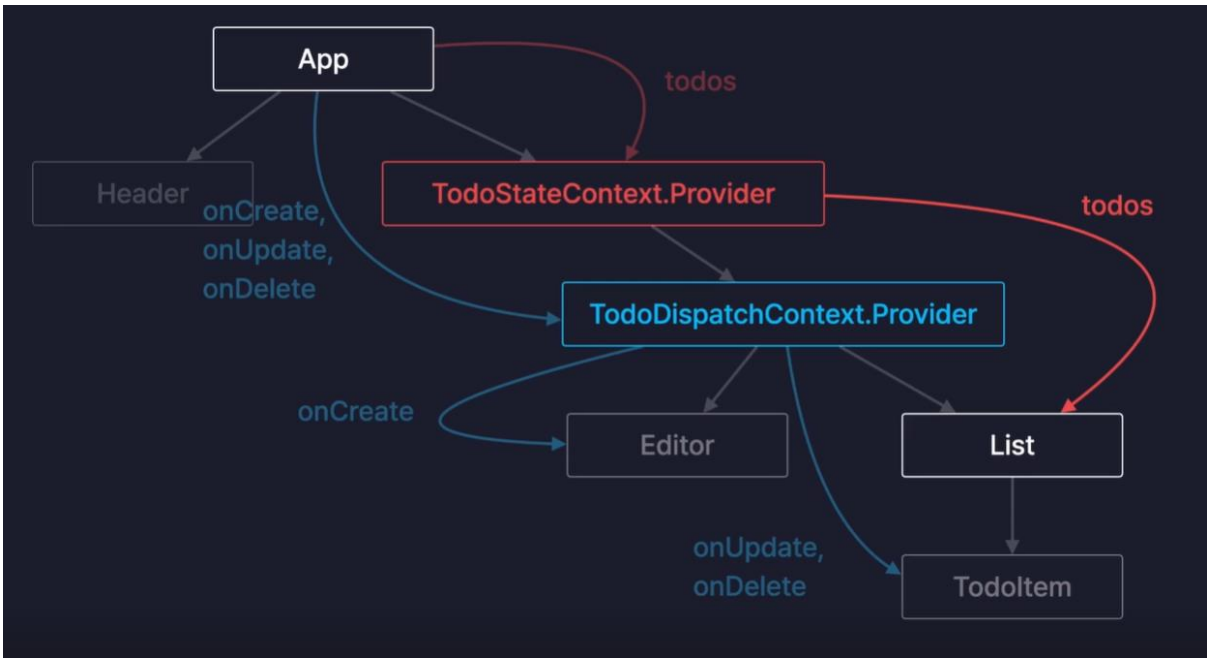
List.jsx수정

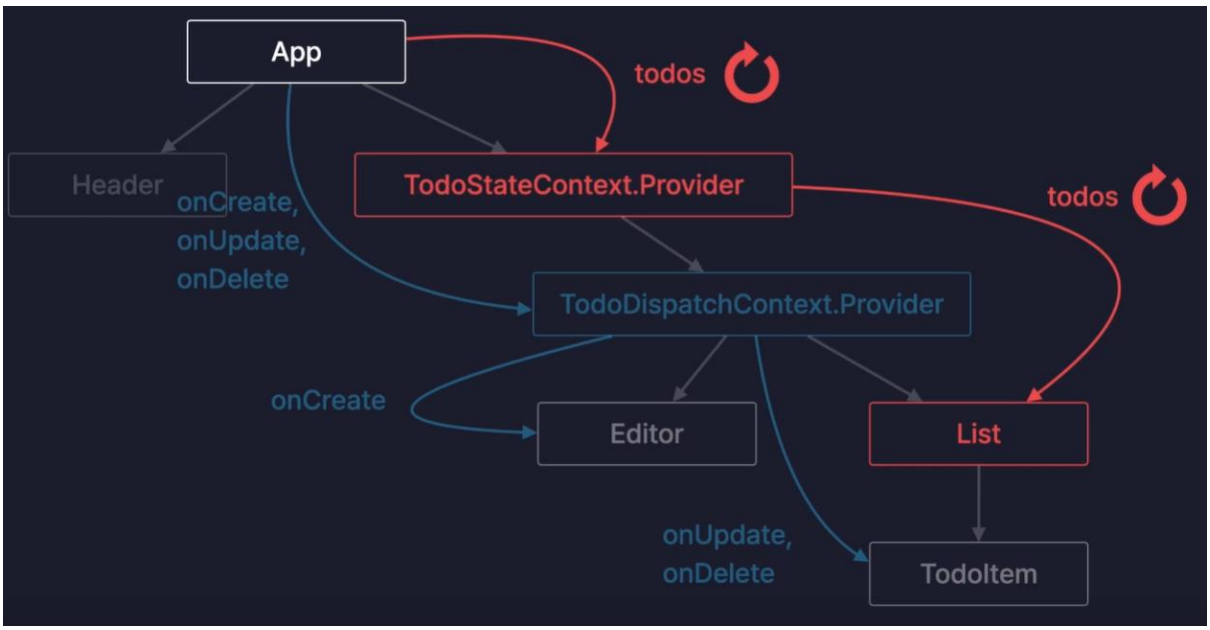
```
const List = () => {

    const todos = useContext(TodoStateContext);
            Provider에서 value를 todos를 그대로 전달 했기 때문에 구조분해 할당 아니다. {} 넣으면 안된다!!
```

TodoTem.jsx수정

```
6   const TodoItem = ({id, isDone, content , date }) => {
7
8       const {onUpdate, onDelete} = useContext(TodoDispatchContext);
```


App
todos
Header
onCreate,
onUpdate,
onDelete
TodoStateContext.Provider
todos
TodoDispatchContext.Provider
onCreate
Editor
List
onUpdate,
onDelete
TodoItem


App
todos
Header
onCreate,
onUpdate,
onDelete
TodoStateContext.Provider
todos
TodoDispatchContext.Provider
onCreate
Editor
List
onUpdate,
onDelete
TodoItem