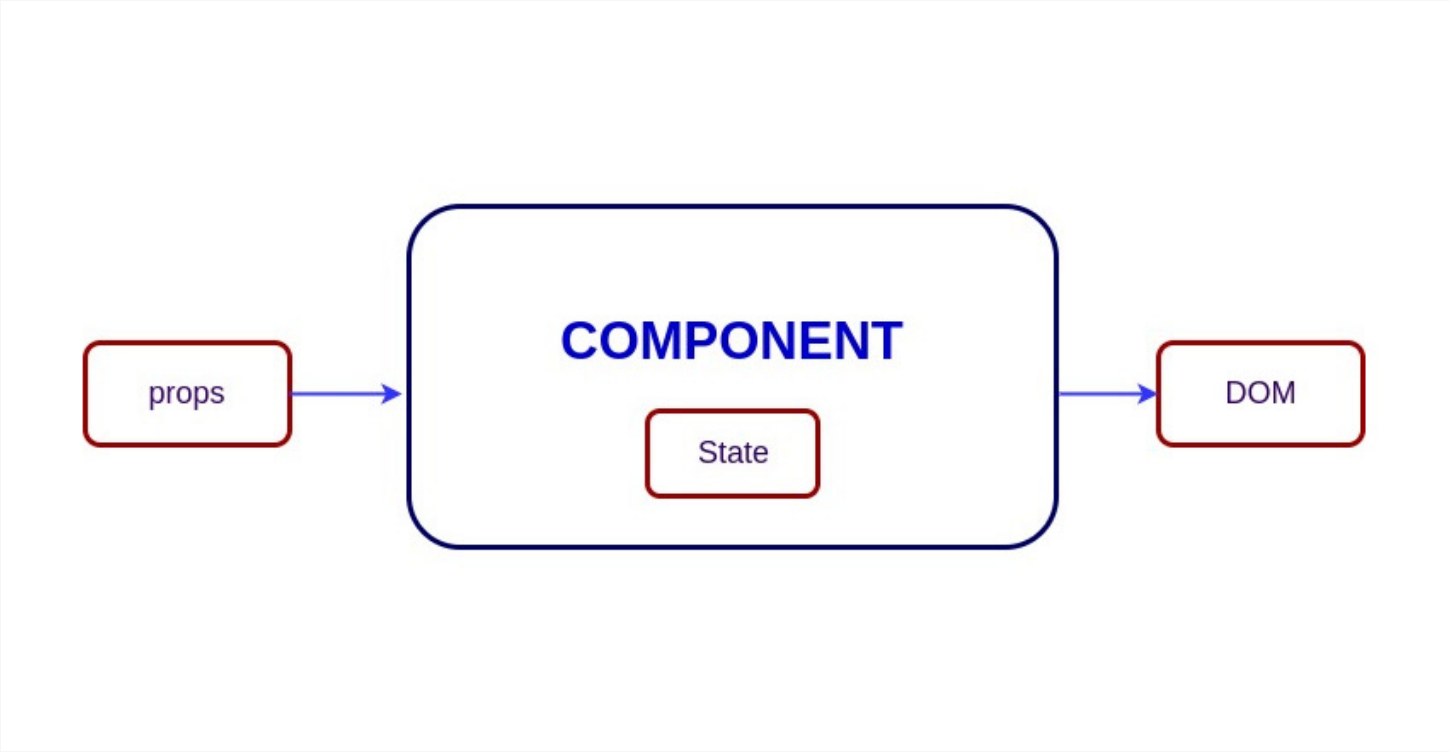
WEB프로젝트

PROPS

state와 useState를 통해 컴포넌트에서 변경되는 값의 상태를 관리 했다면,

props는 다른 컴포넌트에 값(데이터)를 전달할 때 사용한다.



App컴포넌트에서 Hello컴포넌트로 사용자의 이름을 props로 넘겨주고 싶다면 다음과 같이 작성하면 된다.

App.js

```
import Hello from './components/Hello';

const App = () => {
  return (
    <div className="App">
      <Hello name="react"/>
    </div>
  );
}

export default App;
```

Hello.js

```
import React from 'react';

const Hello = (props) => {
  return <div>안녕하세요 {props.name}</div>
}

export default Hello;
```

props는 객체 형태로 전달되며, name 값을 조회하기 위해서는

props.name 과 같은 형태로 props객체의 파라미터name을 조회한다.

다음은 여러 개의 props를 전달하는 방법입니다.

App.js

```
import Hello from './components/Hello';

const App = () => {
  return (
    <div className="App">
      <Hello name="react" color="red"/>
    </div>
  );
}

export default App;
```

Hello.js

```
import React from 'react';

const Hello = (props) => {
    return <div style={{ color: props.color }}>안녕하세요 {props.name}</div>
}

export default Hello;
```

이번에는 Hello 컴포넌트로 전달된 color값을 조회하여, 폰트의 색상을 변경해보았다.

앞과 같은 방식으로 props 값을 조회하기 위해서는 props.를 붙여서 입력을 하고 있지만,
비구조화 할당 방식을 이용하면 코드를 좀 더 간결하게 작성할 수 있다.

Hello.js

```
import React from 'react';

const Hello = (props) => {
    return <div style={{ color: props.color }}>안녕하세요 {props.name}</div>
}

export default Hello;
```

```
import React from 'react';

const Hello = ({color, name}) => {
    return <div style={{ color }}>안녕하세요 { name }</div>
}

export default Hello;
```

(파라미터의 순서는 상관 없다. > { color, name } or { name, color } 둘 다 가능)

(자바스크립트에서 인라인 방식으로 스타일을 지정하고, 스타일 속성 값을 객체로 전달할 때, style의 속성 명과 객체 파라미터 이름이 같다면 속성 명을 생략할 수 있다)

default props를 지정하기.

컴포넌트에 넘겨줄 props값이 지정되지 않았을 때, 기본으로 사용할 값을 설정하고 싶다면
default props를 설정하면 된다.

App.js

```
import Hello from './components/Hello';

const App = () => {
  return (
    <div className="App">
      <Hello name="react" color="red"/>
      <Hello color="pink"/>
    </div>
  );
}

export default App;
```

Hello.js

```
import React from 'react';

const Hello = ({name, color}) => {
    return <div style={{ color }}>안녕하세요 {name}</div>
}

Hello.defaultProps = {
  name: '이름없음'
}

export default Hello;
```

Hello 컴포넌트의 defaultProps { name : '이름없음' } 을 설정했기 때문에
App 컴포넌트에서 color 값만 넘겨준 경우, "안녕하세요 이름없음"이 렌더링 된다.

컴포넌트의 태그 사이에 넣은 값은, props children을 사용하여 조회할 수 있다.

Wrapper.js

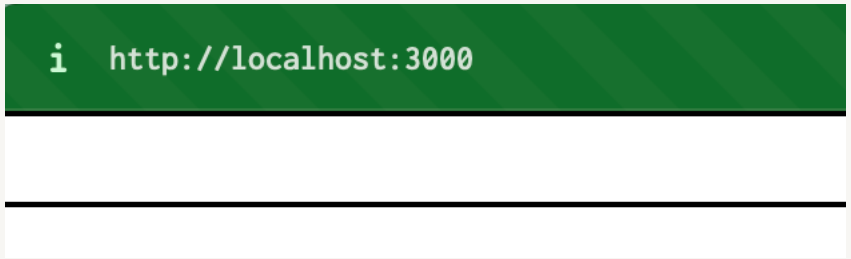
```
import React from 'react';

const Wrapper = () => {
  const style = {
    border: '2px solid black',
    padding: '16px',
  };
  return (
    <div style={style}>

    </div>
  )
}

export default Wrapper;
```

기존의 App 컴포넌트를 감싸는 Wrapper 컴포넌트를 생성한다.

App 컴포넌트를 다음과 같이 수정, Wrapper 컴포넌트를 추가하여 기존 props 태그를 감싼다.

App.js

```
import React from 'react';
import Hello from './components/Hello';
import Wrapper from './components/Wrapper';

const App = () => {
  return (
      <Wrapper>
        <Hello name="react" color="red"/>
        <Hello color="pink"/>
      </Wrapper>
    );
}

export default App;
```



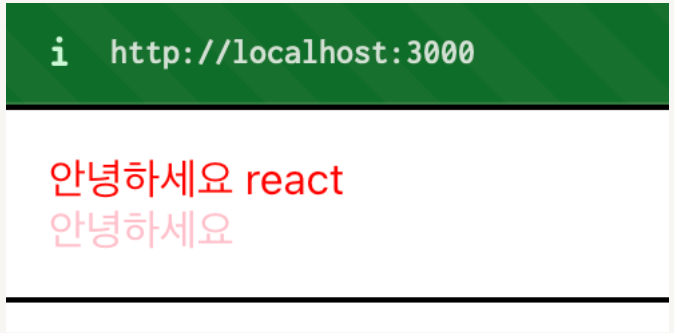
수정 후 리액트 앱을 확인해보면 Wrapper 컴포넌트의 테두리는 출력이 되지만,
안의 props 값은 출력이 되지 않는 것을 볼 수 있다.

Wrapper.js

```jsx
import React from 'react';

const Wrapper = ({children}) => {
  const style = {
    border: '2px solid black',
    padding: '16px',
  };
  return (
    <div style={style}>
        {children}
    </div>
  )
}

export default Wrapper;
```

Wrapper 내부의 내용을 출력하기 위해서는

Wrapper 컴포넌트에서 props.children을 렌더링 해줘야 된다.



PROPS 값에 따른 조건부 렌더링

특정 조건에 따라 다른 화면을 렌더링 하는 것을 조건부 렌더링이라고 한다.

App 컴포넌트에서 Hello 컴포넌트를 사용할 때, isTrue 라는 props를 설정하여 해당 값에 따른 조건부 렌더링을 구현.

```jsx
import React from 'react';
import Hello from './components/Hello';
import Wrapper from './components/Wrapper';

const App = () => {
  return (
    <Wrapper>
      <Hello name="react" color="red" isTrue={true}/>
      <Hello color="pink"/>
    </Wrapper>
  );
}

export default App;
```

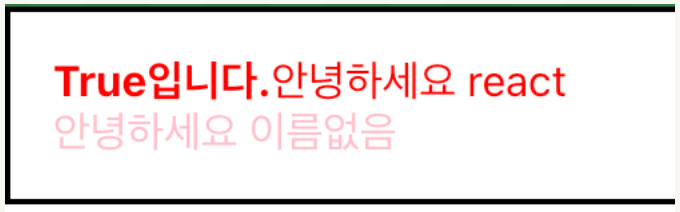
App.js

isTrue 값은 자바스크립트 값이기 때문에 중괄호로 감싼다.

isTrue 값이 true면 "True입니다."를 출력하고 false인 경우, 출력하지 않는다.
해당 기능은 삼항 연산자로 간단하게 구현 가능하다.

```jsx
import React from 'react';

const Hello = ({name, color, isTrue}) => {
    return <div style={{ color }}>
        { isTrue ? <b>True입니다.</b> : null }
        안녕하세요 {name}
        </div>
}

Hello.defaultProps = {
    name: '이름없음'
  }

export default Hello;
```

Hello.js



isTrue 값이 true일 경우, True입니다.를 보여주고, false일 경우, null을 보여준다
(JSX에서 null, false, undefined를 렌더링 한다면 아무것도 나타나지 않는다)

조건부 렌더링의 경우 A or B 다른 내용을 출력하는 경우 삼항 연산자를 사용하지만,
같은 내용을 "보여준다 or 감춘다" 의 경우 && 연산자를 이용하여 간단하게 처리할 수 있다.

```jsx
import React from 'react';

const Hello = ({name, color, isTrue}) => {
    return <div style={{ color }}>
        {isTrue && <b>True입니다.</b>}
        안녕하세요 {name}
        </div>
}

Hello.defaultProps = {
    name: '이름없음'
  }

export default Hello;
```

Hello.js

{ isTrue && True입니다. } 의 경우 전과 마찬가지로 isTrue가 true일 경우, "True입니다."를 출력하고, false일 경우 출력하지 않는 기능을 하지만, 좀 더 간단하게 표현이 가능하다.

TODO LIST

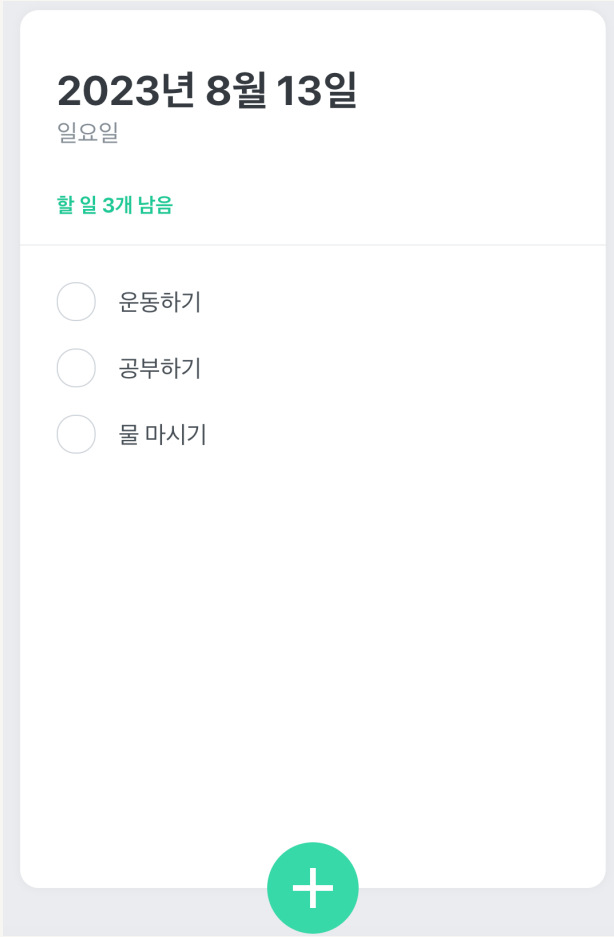
React의 주요 개념들을 이용한 TODO LIST를 제작한다.

각 요소를 나눠서 컴포넌트로 만들기

CSS in JS 작성 기법

복합적인 컴포넌트 구조에 따른 state(상태)관리,

props 처리 방식 등



Head

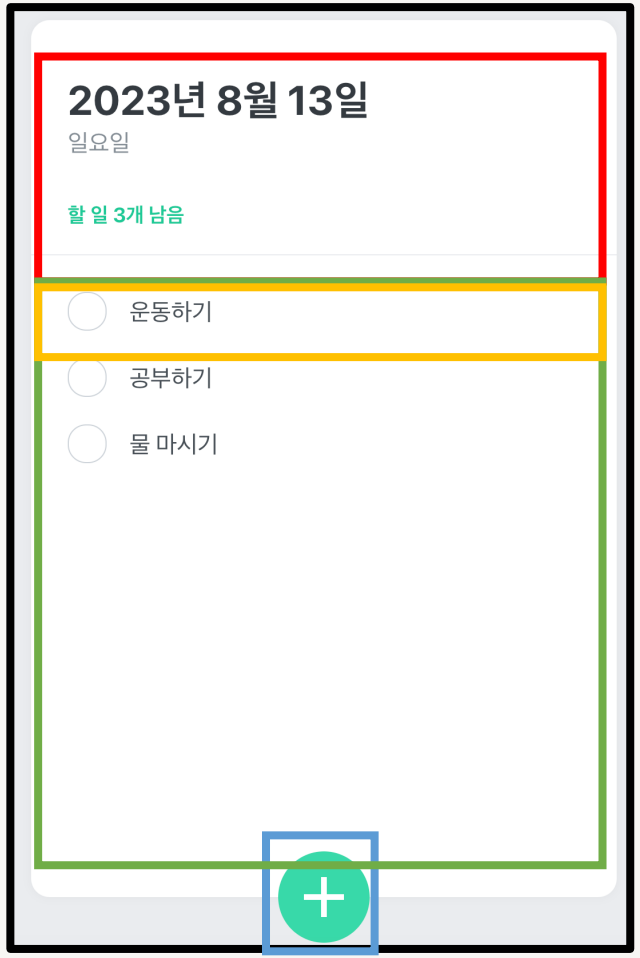
오늘의 날짜, 요일
남은 할 일의 수를 알려준다

List

Item의 내용들을 배열에 저장하여
여러개의 Item 컴포넌트를
렌더링한다

Item

할 일에 대한 정보를 렌더링하며
일의 완료 여부, 삭제 기능을 담당한다



Template

전체 레이아웃을 나타낸다
(흰색 박스)

Create

새로운 할 일을 등록한다.
초록색 원을 렌더링 하고
버튼을 누르면 할 일을 입력하
는 입력 폼이 나타난다.

App.js

```
import React from 'react';
import { createGlobalStyle } from 'styled-components';

const GlobalStyle = createGlobalStyle`
  body {
    background: #e9ecef;
  }
`;

const App = () => {
  return (
    <>
      <GlobalStyle />
      <div>안녕하세요</div>
    </>
  );
}

export default App;
```

App컴포넌트를 수정하여 todolist의 배경색을 지정한다.

styled-components 방식으로 별도의 css파일을 작성하는 것이 아니라, js파일에 스타일을 함께 작성한다.

특정 컴포넌트가 아닌 글로벌 스타일을 지정하는 경우 (위의 경우 div태그가 아닌 body태그에 스타일 적용)

createGlobalStyle를 사용한다.

Components/Template.js

```
import React from 'react';
import styled from 'styled-components';

const TemplateBlock = styled.div`
  width: 512px;
  height: 768px;

  position: relative; /* 추후 박스 하단에 추가 버튼을 위치시키기 위한 설정 */
  background: white;
  border-radius: 16px;
  box-shadow: 0 0 8px 0 rgba(0, 0, 0, 0.04);

  margin: 0 auto; /* 페이지 중앙에 나타나도록 설정 */

  margin-top: 96px;
  margin-bottom: 32px;
  display: flex;
  flex-direction: column;
`;

const Template = ({ children }) => {
  return <TemplateBlock>{children}</TemplateBlock>;
}

export default Template;
```

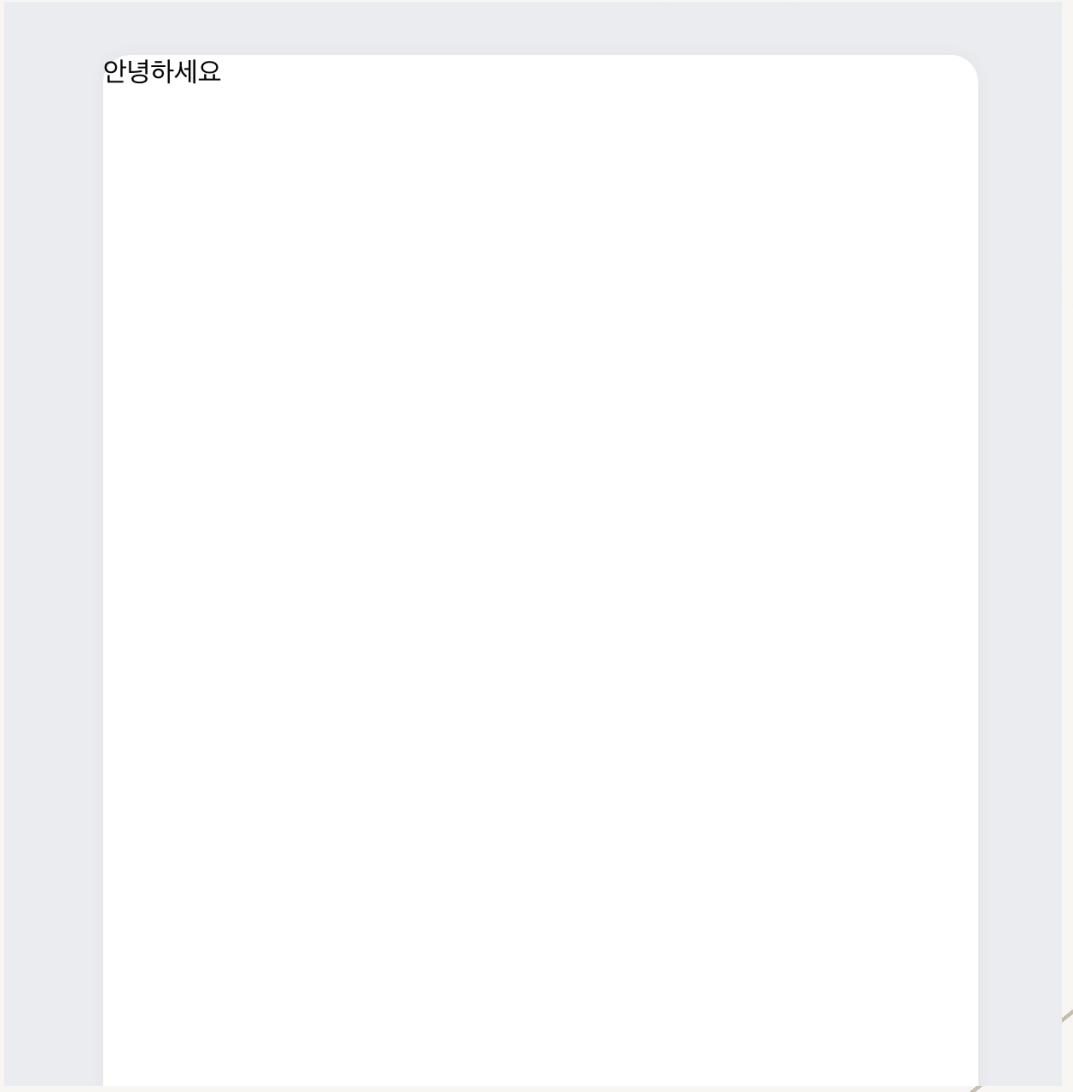
Template컴포넌트를 만들어서
Todolist의 내용이 표시될 하얀 박스를 만든다

마찬가지로 styled-components로,
스타일을 적용하여 박스를 생성한다.
(styled-components가 아닌
css파일을 생성하여 만드는 CSS-in-CSS
방식으로 구현해도 동일하게 구현 가능)

App.js

```jsx
import React from 'react';
import { createGlobalStyle } from 'styled-components';
import Template from './components/Template';

const GlobalStyle = createGlobalStyle`
  body {
    background: #e9ecef;
  }
`;

const App = () => {
  return (
    <>
      <GlobalStyle />
      <Template>안녕하세요</Template>
    </>
  );
}

export default App;
```

Template 컴포넌트를 렌더링한다.



Components/Head.js

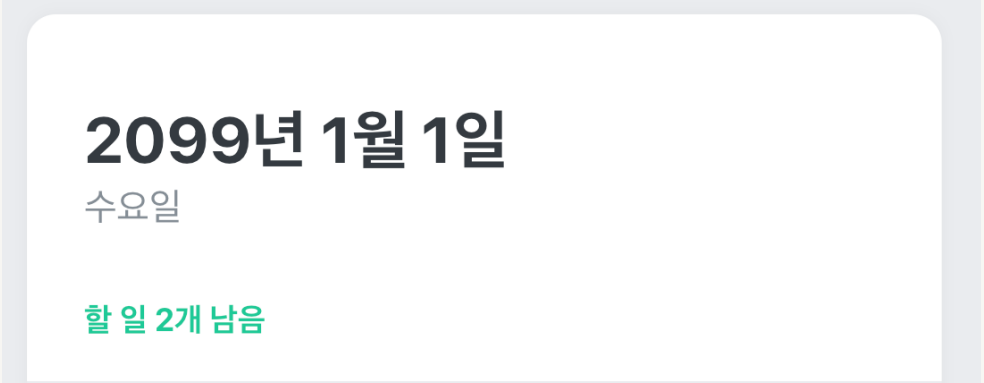
```jsx
import React from 'react';
import styled from 'styled-components';

const HeadBlock = styled.div`
  padding-top: 48px;
  padding-left: 32px;
  padding-right: 32px;
  padding-bottom: 24px;
  border-bottom: 1px solid #e9ecef;
  h1 {
    margin: 0;
    font-size: 36px;
    color: #343a40;
  }
  .day {
    margin-top: 4px;
    color: #868e96;
    font-size: 21px;
  }
  .tasks-left {
    color: #20c997;
    font-size: 18px;
    margin-top: 40px;
    font-weight: bold;
  }
`;

const Head = () => {
  return (
    <HeadBlock>
      <h1>2099년 1월 1일</h1>
      <div className="day">수요일</div>
      <div className="tasks-left">할 일 2개 남음</div>
    </HeadBlock>
  );
}

export default Head;
```

Head 컴포넌트에서는 오늘 날짜, 요일, 남은 할 일의 수를 보여준다.

해당 컴포넌트의 <HeadBlock> 태그 안의 <h1>, 요소들도
별도의 컴포넌트로 만들 수 있지만, 크게 구분이 되거나
특출난 기술이 적용된 부분이 아니기 때문에
하나의 컴포넌트에 HTML태그로 작성하는 방식을 택함.

App.js

```
import React from 'react';
import { createGlobalStyle } from 'styled-components';
import Template from './components/Template';
import Head from './components/Head';

const GlobalStyle = createGlobalStyle`
  body {
    background: #e9ecef;
  }
`;

const App = () => {
  return (
    <>
      <GlobalStyle />
      <Template>
        <Head/>
      </Template>
    </>
  );
}

export default App;
```



Head 컴포넌트를 추가한다

Components/List.js

```
import React from 'react';
import styled from 'styled-components';

const ListBlock = styled.div`
  flex: 1;
  padding: 20px 32px;
  padding-bottom: 48px;
  overflow-y: auto;
  background: gray; /* 사이즈를 확인하기 위한 임시 스타일 (배경색 구분) */
`;

const TodoList = () => {
  return <ListBlock>TodoList</ListBlock>;
}

export default TodoList;
```

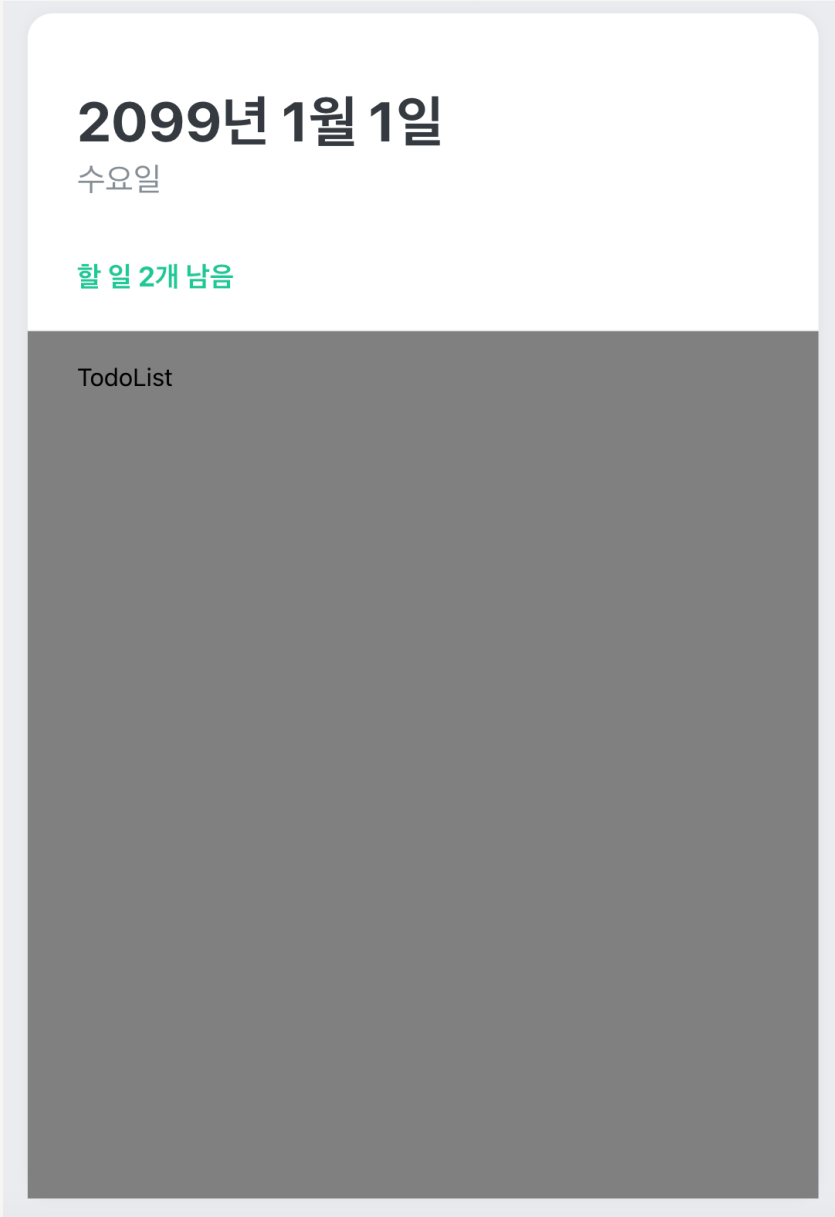
할 일 목록을 출력하는 List 컴포넌트.

아직 출력할 Item 컴포넌트를 생성하지 않았기에 임시로 사이즈만 지정하고 배경색으로 적용 여부를 확인한다.

(flex:1 속성으로 해당 div 영역을 꽉 채우도록 설정한다)

App.js

```
import React from 'react';
import { createGlobalStyle } from 'styled-components';
import Template from './components/Template';
import Head from './components/Head';
import List from './components/List';

const GlobalStyle = createGlobalStyle`
  body {
    background: #e9ecef;
  }
`;

const App = () => {
  return (
    <>
      <GlobalStyle />
      <Template>
        <Head/>
        <List/>
      </Template>
    </>
  );
}

export default App;
```



List 컴포넌트를 추가한다.

List 영역이 설정되어 배경색이 변경되었다.

Components/Item.js

```
import React from 'react';
import styled, { css } from 'styled-components';
import { MdDone, MdDelete } from 'react-icons/md';

const Remove = styled.div`
  display: flex;
  align-items: center;
  justify-content: center;
  color: #dee2e6;
  font-size: 24px;
  cursor: pointer;
  &:hover {
    color: #ff6b6b;
  }
  display: none;
`;

const TodoItemBlock = styled.div`
  display: flex;
  align-items: center;
  padding-top: 12px;
  padding-bottom: 12px;
  &:hover {
    ${Remove} {
      display: initial;
    }
  }
`;
```

```
const CheckCircle = styled.div`
  width: 32px;
  height: 32px;
  border-radius: 16px;
  border: 1px solid #ced4da;
  font-size: 24px;
  display: flex;
  align-items: center;
  justify-content: center;
  margin-right: 20px;
  cursor: pointer;
  ${props =>
    props.done &&
    css`
      border: 1px solid #38d9a9;
      color: #38d9a9;
    `}
`;

const Text = styled.div`
  flex: 1;
  font-size: 21px;
  color: #495057;
  ${props =>
    props.done &&
    css`
      color: #ced4da;
    `}
`;

const Item = ({ id, done, text }) => {
  return (
    <ItemBlock>
      <CheckCircle done={done}>{done && <MdDone />}</CheckCircle>
      <Text done={done}>{text}</Text>
      <Remove>
        <MdDelete />
      </Remove>
    </ItemBlock>
  );
}

export default Item;
```

```
import { MdDone, MdDelete } from 'react-icons/md';
```

react-icons에서 제공하는 아이콘을 사용하기 위해 해당 라이브러리를 설치한다.

`**npm install react-icons**`

```
const ItemBlock = styled.div`
  display: flex;
  align-items: center;
  padding-top: 12px;
  padding-bottom: 12px;
  &:hover {
    ${Remove} {
      display: initial;
    }
  }
`;
```

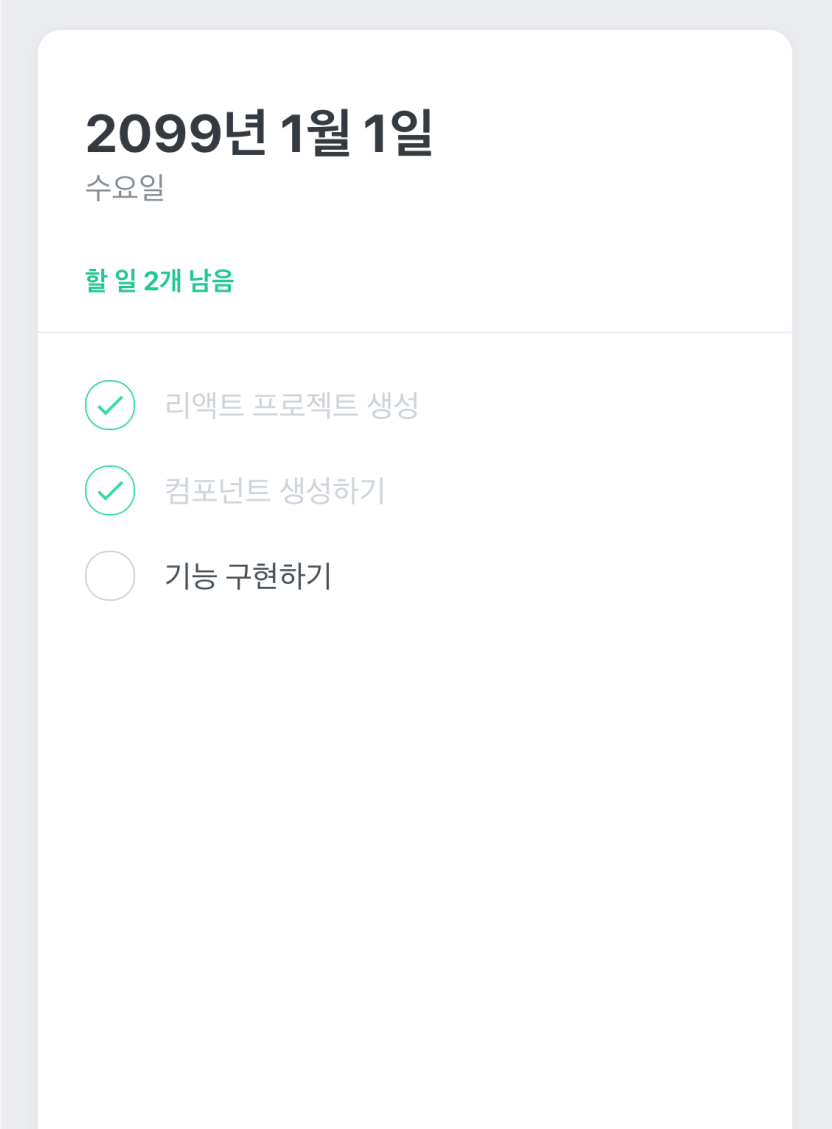
해당 스타일은 components selector로
ItemBlock위에 커서가 있을 때, Remove 컴포넌트를
보여준다는 의미를 가진다.

```
const CheckCircle = styled.div`
  width: 32px;
  height: 32px;
  border-radius: 16px;
  border: 1px solid #ced4da;
  font-size: 24px;
  display: flex;
  align-items: center;
  justify-content: center;
  margin-right: 20px;
  cursor: pointer;
  ${props =>
    props.done &&
    css`
      border: 1px solid #38d9a9;
      color: #38d9a9;
    `}
`;
```

CheckCirlce 컴포넌트가 전달받은 props.done의 값
true일 경우, 아래의 css 스타일을 적용시킨다.

List.js

```jsx
import React from 'react';
import styled from 'styled-components';
import Item from './Item';

const ListBlock = styled.div`
  flex: 1;
  padding: 20px 32px;
  padding-bottom: 48px;
  overflow-y: auto;
`;

const List = () => {
  return (
    <ListBlock>
      <Item text="리액트 프로젝트 생성" done={true} />
      <Item text="컴포넌트 생성하기" done={true} />
      <Item text="기능 구현하기" done={false} />
    </ListBlock>
  )
}

export default List;
```



List 컴포넌트에 출력할 Item 컴포넌트 내용을 추가한다.

(배경색을 지정한 임시 스타일은 제거한다.)

Components/Create.js

```
import React, { useState } from 'react';
import styled, { css } from 'styled-components';
import { MdAdd } from 'react-icons/md';

const CircleButton = styled.button`
  background: #38d9a9;
  &:hover {
    background: #63e6be;
  }
  &:active {
    background: #20c997;
  }

  z-index: 5;
  cursor: pointer;
  width: 80px;
  height: 80px;
  display: block;
  align-items: center;
  justify-content: center;
  font-size: 60px;
  position: absolute;
  left: 50%;
  bottom: 0px;
  transform: translate(-50%, 50%);
  color: white;
  border-radius: 50%;
  border: none;
  outline: none;
  display: flex;
  align-items: center;
  justify-content: center;
  transition: 0.125s all ease-in;
  ${props =>
    props.open &&
    css`
      background: #ff6b6b;
      &:hover {
        background: #ff8787;
      }
      &:active {
        background: #fa5252;
      }
      transform: translate(-50%, 50%) rotate(45deg);
    `}
`;
```

하단의 +버튼을 생성하며,

크기, 위치 정렬, 애니메이션 적용 등

스타일을 적용한다.

Components/Create.js

```
const InsertFormPositioner = styled.div`
  width: 100%;
  bottom: 0;
  left: 0;
  position: absolute;
`;

const InsertForm = styled.form`
  background: #f8f9fa;
  padding-left: 32px;
  padding-top: 32px;
  padding-right: 32px;
  padding-bottom: 72px;

  border-bottom-left-radius: 16px;
  border-bottom-right-radius: 16px;
  border-top: 1px solid #e9ecef;
`;

const Input = styled.input`
  padding: 12px;
  border-radius: 4px;
  border: 1px solid #dee2e6;
  width: 100%;
  outline: none;
  font-size: 18px;
  box-sizing: border-box;
`;
```

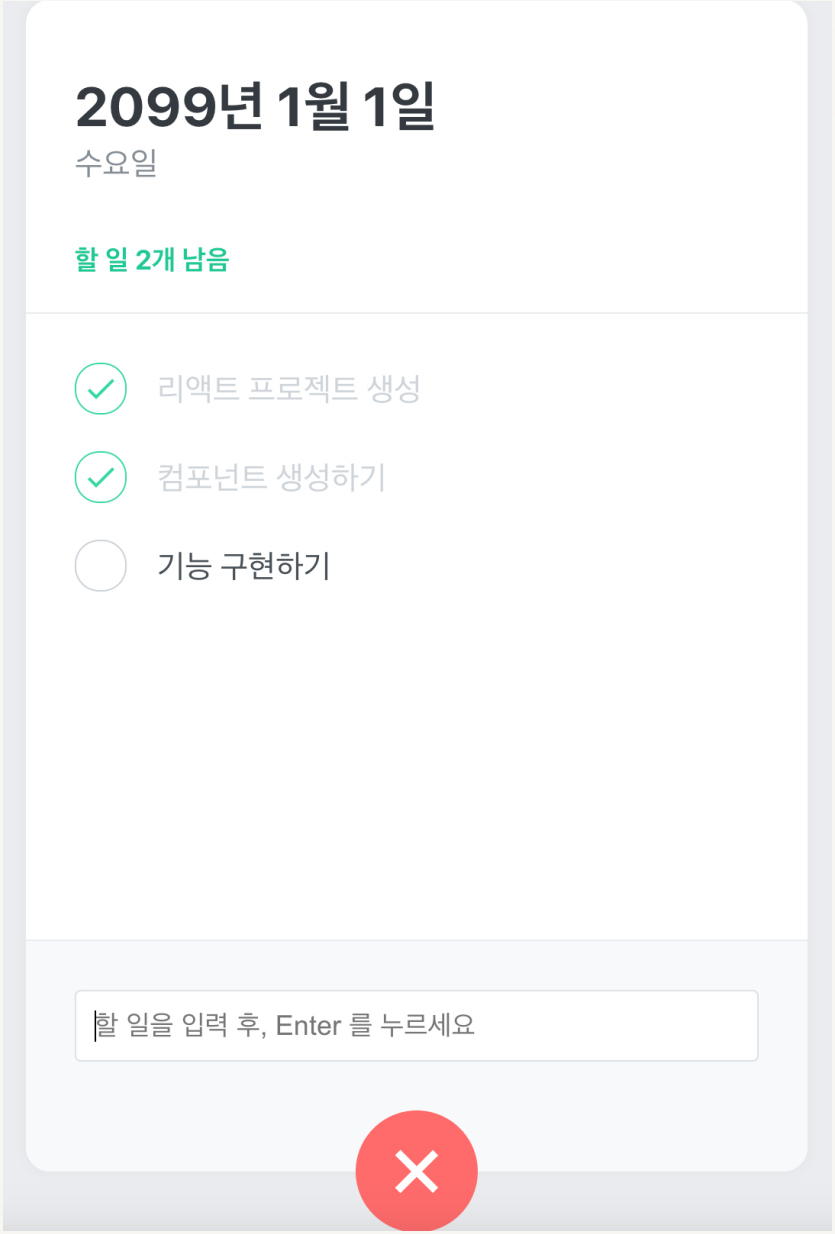
```
const Create = () => {
  const [open, setOpen] = useState(false);

  const onToggle = () => setOpen(!open);

  return (
    <>
      {open && (
        <InsertFormPositioner>
          <InsertForm>
            <Input autoFocus placeholder="할 일을 입력 후, Enter 를 누르세요" />
          </InsertForm>
        </InsertFormPositioner>
      )}
      <CircleButton onClick={onToggle} open={open}>
        <MdAdd />
      </CircleButton>
    </>
  );
}

export default Create;
```

ustState를 사용하여 변경할 수 있는 open 값을 관리한다.

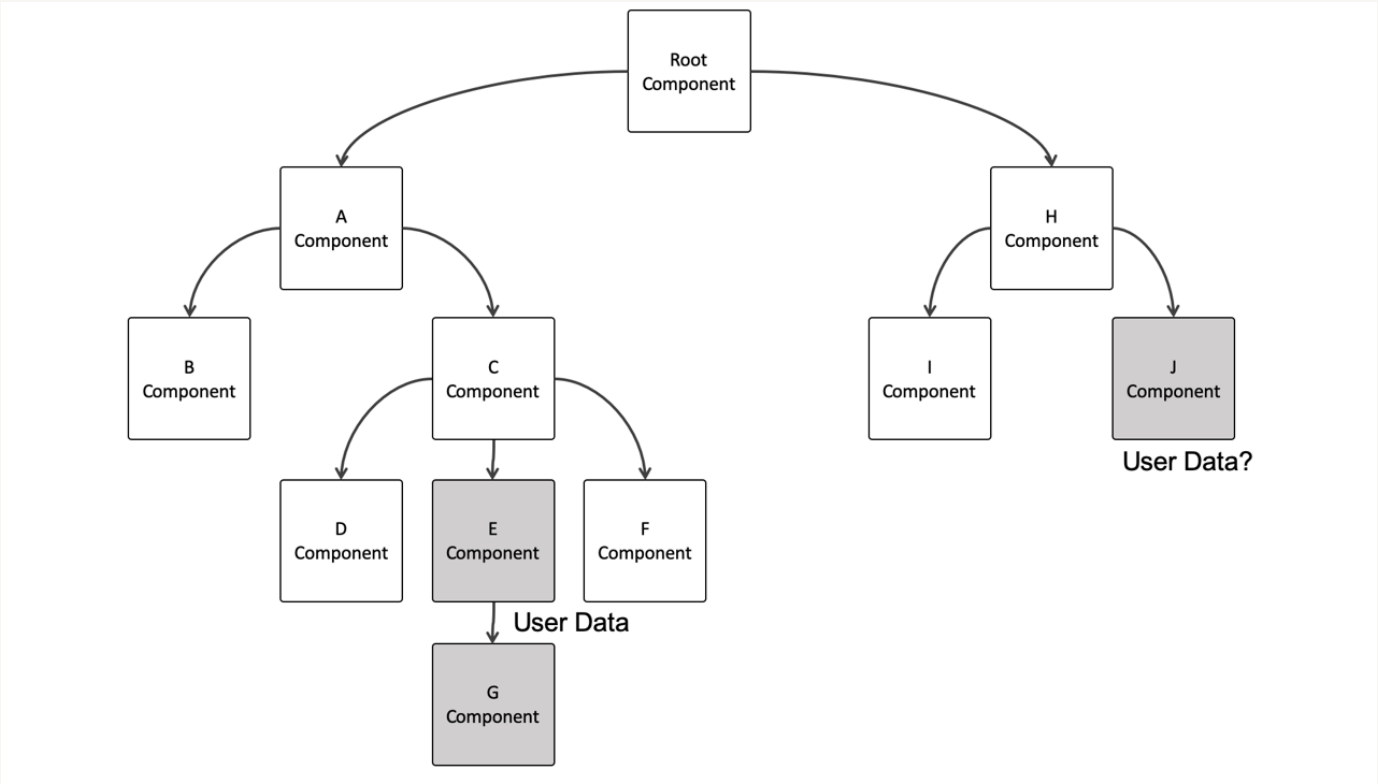
해당 값이 true일 때 아이콘을 45도 돌려서 X 모양으로 보이게 하고,
버튼의 색을 빨간색으로 변경한다.
그 동시에 사용자가 입력할 수 있는 입력 폼도 나타낸다.

App.js

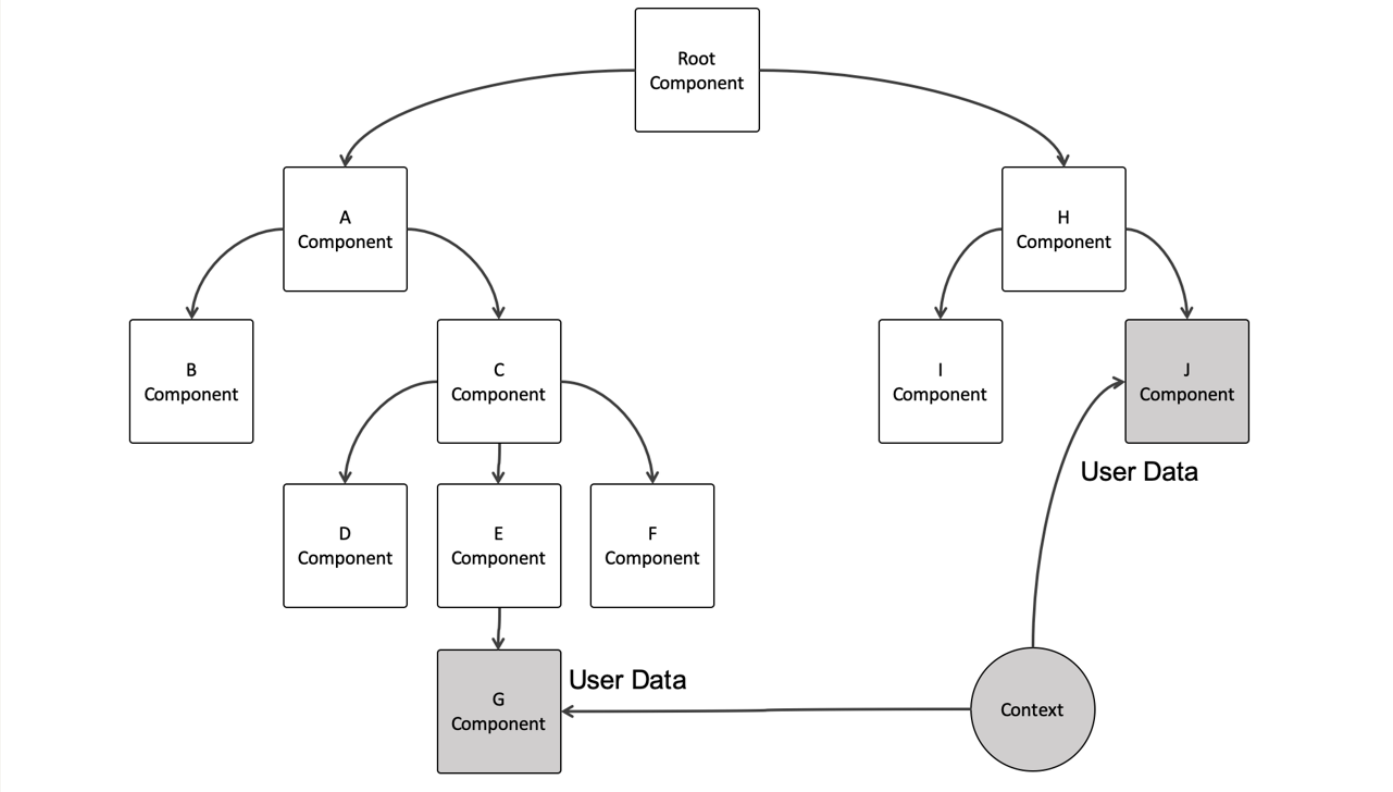
```
import React from 'react';
import { createGlobalStyle } from 'styled-components';
import Template from './components/Template';
import Head from './components/Head';
import List from './components/List';
import Create from './components/Create'

const GlobalStyle = createGlobalStyle`
  body {
    background: #e9ecef;
  }
`;

const App = () => {
  return (
    <>
      <GlobalStyle />
      <Template>
        <Head/>
        <List/>
        <Create/>
      </Template>
    </>
  );
}

export default App;
```



CONTEXT API


Root Component
A Component
H Component
B Component
C Component
I Component
J Component
User Data?
D Component
E Component
F Component
User Data
G Component

CONTEXT API



전역 데이터를 Context라는 공간에 저장하고 관리하면 불필요하게 props를 주고 받는 과정을 줄일 수 있음.

TodoContext.js

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

const initialTodos = [
  {
    id: 1,
    text: '프로젝트 생성하기',
    done: true
  },
  {
    id: 2,
    text: '컴포넌트 스타일링하기',
    done: true
  },
  {
    id: 3,
    text: '기능 구현하기',
    done: false
  }
];

const todoReducer = (state, action) => {
  switch (action.type) {
    case 'CREATE':
      return state.concat(action.todo);
    case 'TOGGLE':
      return state.map(todo =>
        todo.id === action.id ? { ...todo, done: !todo.done } : todo
      );
    case 'REMOVE':
      return state.filter(todo => todo.id !== action.id);
    default:
      throw new Error(`Unhandled action type: ${action.type}`);
  }
}

export const TodoProvider = ({ children }) => {
  const [state, dispatch] = useReducer(todoReducer, initialTodos);
  return children;
}
```

TodoContext.js 파일을 생성하고,

useReducer를 사용하여 상태를 관리하는

TodoProvider라는 컴포넌트를 생성한다.

useReducer
(복잡한 상태 관리를 도와주는 기능)

TodoContext.js

```
import React, { useReducer, createContext } from 'react';
```

```
const TodoStateContext = createContext();
const TodoDispatchContext = createContext();

export const TodoProvider = ({ children }) => {
    const [state, dispatch] = useReducer(todoReducer, initialTodos);
    return (
      <TodoStateContext.Provider value={state}>
        <TodoDispatchContext.Provider value={dispatch}>
          {children}
        </TodoDispatchContext.Provider>
      </TodoStateContext.Provider>
    );
  }
```

state와 dispatch*를
(Reducer에 액션, 이벤트를 전달하여 상태를 변경할 수 있게 한다)

다른 컴포넌트에서 바로 사용할 수 있도록,

각각의 Context를 생성한다.

해당 컴포넌트에서 useContext를 사용해 컴포넌트간의 상태를 공유할 수 있지만, 커스텀 Hook을 만들어서 useContext의 기능을 사용하는 방식으로 구현한다.

Hook이란

컴포넌트의 실행 단계와 관련된 생명주기 메서드, 상태 관리(state) 등을 더 효율적으로 할 수 있도록 도움을 주는 기능이다.

커스텀Hook은 개발자가 자신만의 기능이나 로직을 재사용 할 수 있는 형태로 생산한 함수를 뜻한다.

TodoContext.js

```
import React, { useReducer, createContext, useContext } from 'react';
```

useContext 추가

```
50  export const useTodoState = () => {
51    return useContext(TodoStateContext);
52  }
53
54  export const useTodoDispatch = () => {
55    return useContext(TodoDispatchContext);
56  }
```

각각 state와 dispatch를 전달하는 커스텀Hook

useTodoState와
useTodoDispatch를 생성한다.

TodoContext.js

할 일 목록의 수를 카운트 하기 위한 고유 ID 값을 넘겨주기 위한 Context를 추가한다.

```
import React, { useReducer, createContext, useContext, useRef } from 'react';
```

useRef 추가

```
36  const TodoStateContext = createContext();
37  const TodoDispatchContext = createContext();
38  const TodoNextIdContext = createContext();
39
40  export const TodoProvider = ({ children }) => {
41    const [state, dispatch] = useReducer(todoReducer, initialTodos);
42    const nextId = useRef(5);
43    return (
44      <TodoStateContext.Provider value={state}>
45        <TodoDispatchContext.Provider value={dispatch}>
46          <TodoNextIdContext.Provider value={nextId}>
47            {children}
48          </TodoNextIdContext.Provider>
49        </TodoDispatchContext.Provider>
50      </TodoStateContext.Provider>
51    );
52  }
```

useRef

DOM요소나 다른 컴포넌트의 상태 혹은 그 외의 값을 참조하는 데 사용된다.

```
export const useTodoState = () => {
  const context = useContext(TodoStateContext);
  if (!context) {
    throw new Error('Cannot find TodoProvider');
  }
  return context;
}
```

```
63  export const useTodoNextId = () => {
64    return useContext(TodoNextIdContext);
65  }
```

커스텀 hook을 다음과 같이 수정해주면
문제가 발생한 경우 메세지가 발생하기 때문에
추후 오류 수정 및 유지보수에 용이하다.
(필수는 아님)

App.js

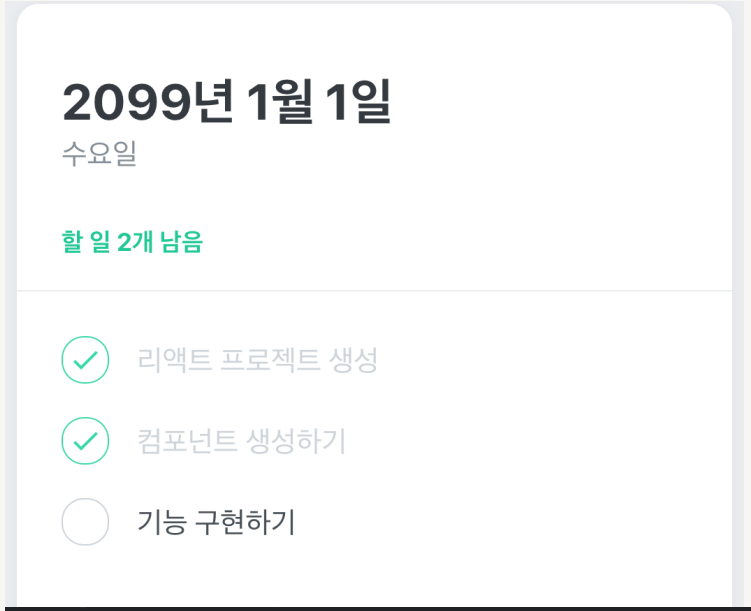
```
import React from 'react';
import { createGlobalStyle } from 'styled-components';
import Template from './components/Template';
import Head from './components/Head';
import List from './components/List';
import Create from './components/Create'
import { TodoProvider } from './TodoContext';

const GlobalStyle = createGlobalStyle`
  body {
    background: #e9ecef;
  }
`;

const App = () => {
  return (
    <TodoProvider>
      <GlobalStyle />
      <Template>
        <Head/>
        <List/>
        <Create/>
      </Template>
    </TodoProvider>
  );
}

export default App;
```

앞에서 만든 관련 Context들을 적용하기 위해

App 컴포넌트의 모든 내용을 TodoProvider로 감싸준다.

components/Head.js

```jsx
import React from 'react';
import styled from 'styled-components';
import { useTodoState } from '../TodoContext';

const HeadBlock = styled.div`
  padding-top: 48px;
  padding-left: 32px;
  padding-right: 32px;
  padding-bottom: 24px;
  border-bottom: 1px solid #e9ecef;
  h1 {
    margin: 0;
    font-size: 36px;
    color: #343a40;
  }
  .day {
    margin-top: 4px;
    color: #868e96;
    font-size: 21px;
  }
  .tasks-left {
    color: #20c997;
    font-size: 18px;
    margin-top: 40px;
    font-weight: bold;
  }
`;

const Head = () => {
    const todos = useTodoState();
    console.log(todos);
  return (
    <HeadBlock>
      <h1>2099년 1월 1일</h1>
      <div className="day">수요일</div>
      <div className="tasks-left">할 일 2개 남음</div>
    </HeadBlock>
  );
}

export default Head;
```




```
▼ (3) [{…}, {…}, {…}] i
  ▶ 0: {id: 1, text: '프로젝트 생성하기', done: true}
  ▶ 1: {id: 2, text: '컴포넌트 스타일링하기', done: true}
  ▶ 2: {id: 3, text: '기능 구현하기', done: false}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

최종적으로 Head 컴포넌트에서 useTodoState로 Context가 가지고 있는 할 일 목록의 state를 콘솔에서 확인할 수 있다.