



# 5\_Context API

## Context API

- React 컴포넌트 트리 전체에서 데이터를 공유할 수 있는 방법
- Props Drilling, State Lifting Up 을 사용하지 않고 부모, 자식 컴포넌트 데이터 전달 가능 (Props Drilling, State Lifting Up 을 반복적으로 작성하는 문제 해결)

## Context 사용법

### [React.]createContext() 함수 : Context 객체 생성

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

const Context변수명 = createContext(); // Context 객체 생성
```

### [React.]useContext() 함수 : 지정된 Context 객체에 담긴 값을 얻어옴

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

const 변수명 = useContext(Context명); // 지정된 Context 객체에 담긴 값을 얻어옴
```

### <Context명.Provider value={값}> : 하위 컴포넌트에게 값을 제공

(1단계 하위 컴포넌트 뿐만 아니라 모든 하위 컴포넌트에 value가 제공됨)

```
<Context명.Provider value={값}>
  <하위 컴포넌트/> </* 하위 컴포넌트 이하 모든 컴포넌트에 value 값 제공*/>
</Context명.Provider>
```

# Context 예제 1 (하위 컴포넌트로 전달)

## 1. src/components/R10\_Context1.js 생성 후 작성

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

/* 1. Context 객체 생성 */
const TestContext = createContext();

/* 4. 손주 (2단계 하위) 컴포넌트 */
const GrandChild = () => {

  // 현재 컴포넌트에서 Context 객체를 사용
  // -> TestContext에 담긴 값을 얻어와 temp에 저장
  const temp = useContext(TestContext);

  return(
    <>
      { /* Parent에서 전달한 값이 Child를 거치지 않고 바로 GrandChild로 전달된 형태 */ }
      <h3>GrandChild Component ({temp}) </h3>
    </>
  );
}

/* 3. 자식 (1단계 하위) 컴포넌트 */
const Child = () => {
  return(
    <>
      <h2>Child Component</h2>
      <GrandChild/>
    </>
  );
}

/* 2. 부모 컴포넌트 */
const Parent = () => {

  return(
    /* Context 객체를 이용해 하위 컴포넌트에 value 제공 */
    <TestContext.Provider value='Parent에서 전달한 값'>
      <h1>Parent Component</h1>

      <Child/> { /* Child 포함 모든 하위 컴포넌트에서 Parent가 제공한 value 사용 가능 */ }
    </TestContext.Provider>
  );
}

export default Parent;
```

## 2. src/App.js 작성

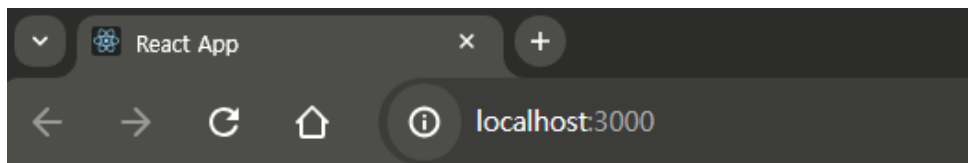
```
import './App.css';
import Context1 from './components/R10_Context1'

function App() {
  return (

    <>
      <Context1 />
    </>
  );
}

export default App;
```

## 3. 결과 화면



**Parent Component**

**Child Component**

**GrandChild Component (Parent에서 전달한 값)**

---

## Context 예제 2(상위 컴포넌트로 전달)

### 1. src/components/R11\_Context2.js 생성 후 작성

```

import React, { useState, createContext, useContext } from 'react';

/* 1. Context 객체 생성 */
const TestContext = createContext();

/* 4. 손주 (2단계 하위) 컴포넌트 */
const GrandChild = () => {

  // 현재 컴포넌트에서 Context 객체를 사용
  // -> TestContext에 담긴 값(객체)을 얻어와 key가 일치하는 변수에 저장
  const { number, setNumber } = useContext(TestContext);

  return(
    <>
      <h3>GrandChild Component</h3>

      {/*
        input의 값이 변할 때 마다 setNumber() 함수 호출
        -> setNumber() == Parent 컴포넌트의 상태 변수
        -> Parent 상태 변수의 값이 변함 == 하위 컴포넌트에서 상위 컴포넌트로 값 전달
      */}
      <input type='number' onChange={ (e) => { setNumber(e.target.value)}} value={number}/>
    </>
  );
}

/* 3. 자식 (1단계 하위) 컴포넌트 */
const Child = () => {
  return(
    <>
      <h2>Child Component</h2>
      <GrandChild/>
    </>
  );
}

/* 2. 부모 컴포넌트 */
const Parent = () => {

  // 상태 변수 선언
  const [number, setNumber] = useState(0);

  return(

    /* Context 객체를 이용해 하위 컴포넌트에 number, setNumber (객체 형태) 제공 */
    /* {"number" : number , "setNumber" : setNumber}
    <TestContext.Provider value={ {number, setNumber} } >
      <h1>Parent Component</h1>

      {/* 상태 변수 number 값을 출력 (GrandChild 에서 상태 값 변경) */}
      <h1>GrandChild에서 전달한 값 : <span className='color-red'>{number}</span></h1>

      <Child /> {/* Child 포함 모든 하위 컴포넌트에서 Parent가 제공한 value 사용 가능 */}
    </TestContext.Provider>
  );
}

```

```
}  
  
export default Parent;
```

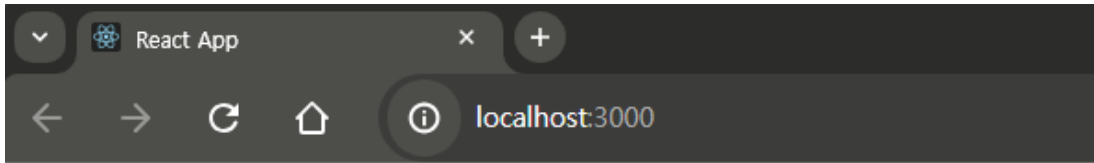
## 2. src/App.js 작성

```
import './App.css';  
import Context2 from './components/R11_Context2'  
  
function App() {  
  return (  
  
    <>  
      <Context2/>  
    </>  
  );  
}  
  
export default App;
```

## 3. App.css 코드 추가

```
.color-red {color: red;}
```

## 4. 결과 화면



## Parent Component

GrandChild에서 전달한 값 : 3

## Child Component

### GrandChild Component

## Context 예제 3(분리된 파일 형태에서 사용법)

### 1. src/components/R13\_Context/UserContext.js 생성 후 작성

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

/* Context 객체 생성 */
const UserContext = createContext();

/* 외부 import 시 내보내기할 Context 객체 지정 */
export default UserContext;
```

### 2. src/components/R13\_Context/Parent.js 생성 후 작성

```

import React, { useState } from 'react';

/* 외부 Context 객체 import */
import UserContext from './UserContext';

/* 외부 Child 컴포넌트 import */
import Child from './Child';

/* 부모 컴포넌트 */
const Parent = () => {

  /* 상태 변수 userList */
  const [userList, setUserList] = useState([]);

  return(

    /* 하위 컴포넌트에 Context를 이용해서 userList, setUserList 제공 */
    <UserContext.Provider value={ {userList, setUserList} }>
      <Child/>

      {/* 상태 변수 userList에 저장된 값을 화면에 출력 */}
      <div>
        {/* key 속성 : 배열(list) 출력 시 요소를 구분하는 key값 */}

        {userList.map( (user, index) => {
          return(
            <ul key={index}>
              <li>index : {index}</li>
              <li>name : {user.name}</li>
              <li>age : {user.age}</li>
            </ul>);
          } )}
        </div>

      </UserContext.Provider>
    );
  }

  export default Parent;

```

### 3. src/components/R13\_Context/Child.js 생성 후 작성

```

import React, { useContext, useState } from 'react';

/* 외부 Context 객체 import */
import UserContext from './UserContext';

/* 자식 컴포넌트 */
const Child = () => {

  /* Context를 통해 제공된 값을 얻어와 변수명과, key가 일치하는 변수에 대입 */

```

```

const {userList, setUserList} = useContext(UserContext);

/* 상태 변수 선언 */
const [inputName, setInputName] = useState('');
const [inputAge, setInputAge] = useState('');

/* 추가 버튼 클릭 시 수행할 함수(이벤트 핸들러) */
const addUser = () => {

    // 상태 변수에 저장된 값을 이용해 user 객체 생성
    const user = {name : inputName, age : inputAge};

    // Context를 통해 제공 받은 userList 깊은 복사 + user 객체를 추가한 새로운 배열 생성
    const newUserList = [ ...userList, user];

    // Context를 통해 제공 받은 setUserList()를 이용해 부모의 상태 변수 값을 수정
    setUserList(newUserList);

    // 현재 컴포넌트의 상태 변수를 빈칸으로 수정(내용 지우기)
    setInputName('');
    setInputAge('');
}

return(
    <div>
        <label>이름 : </label>
        <input type='text' onChange={ e => setInputName(e.target.value)} value={inputName}/>

        <br/>

        <label>나이 : </label>
        <input type='number' onChange={ e => setInputAge(e.target.value)} value={inputAge}/>

        <button onClick={addUser}>추가</button>
    </div>
);
}

export default Child;

```

## 4. src/App.js 작성

```

import './App.css';
import Context3 from './components/R13_Context3/Parent'

function App() {
    return (

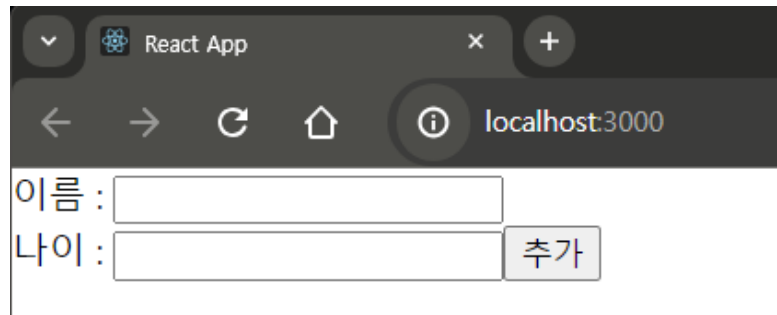
        <>
            <Context3/>
        </>
    )
}

```

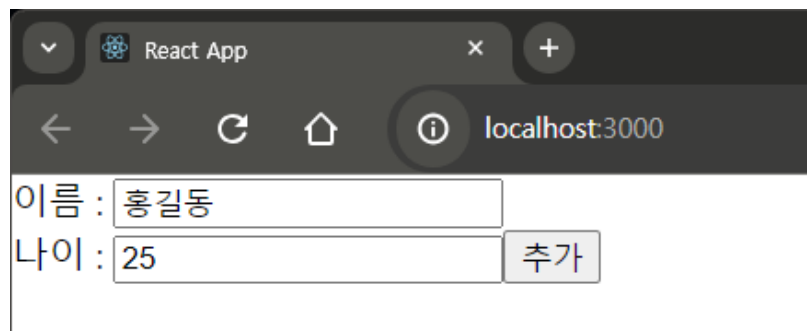


```
);  
}  
  
export default App;
```

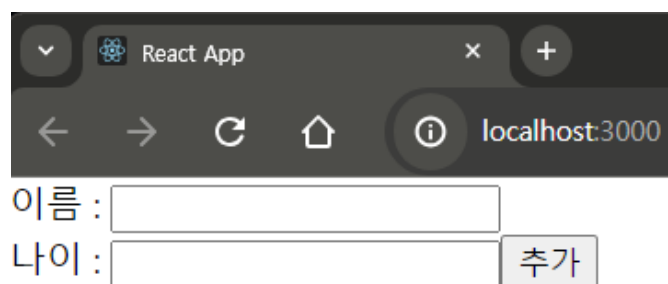
## 5. 결과 화면



A screenshot of a web browser window titled "React App" at the address "localhost:3000". The browser's address bar shows navigation icons and the URL. Below the address bar, there is a form with two input fields. The first field is labeled "이름 :" (Name) and is empty. The second field is labeled "나이 :" (Age) and is also empty. To the right of the age input field is a button labeled "추가" (Add).



A screenshot of the same web browser window. The "이름 :" (Name) input field now contains the text "홍길동" (Hong Gildong). The "나이 :" (Age) input field now contains the number "25". The "추가" (Add) button remains to the right of the age field.



A screenshot of the web browser window showing the form with empty input fields for "이름 :" (Name) and "나이 :" (Age). The "추가" (Add) button is still present.

- index : 0
- name : 홍길동
- age : 25

