**05\_컴포넌트 반복 렌더링**

**1. 리스트 렌더링 기본 개념**

**정의**

웹페이지에서 **게시글, 댓글, 상품, 일정 목록** 등 여러 개의 유사한 데이터를 화면에 표시하는 핵심 기법

**기본 원리**

JavaScript의 map() 함수 + JSX를 결합하여 배열을 컴포넌트로 변환

|  |
| --- |
| **const** items = ['제주', '한려수도', '대관령', '금강산'];  **return** (  <ul>  {items.map(item => <li key={item}>{item}</li>)}  </ul>  ); |

**2. map() 함수 핵심**

**개념**

* 배열의 각 요소를 변환하여 **새로운 배열을 반환**
* **원본 배열을 변경하지 않음**

**React에서 활용 - 두 가지 방식**

**방식 1: 변수에 저장**

|  |
| --- |
| **function** MapRendering1() {  **const** fruits = ["사과", "바나나", "포도"];  **const** fruitsList = fruits.map((fruit, idx) => (  <li key={idx}>{fruit}</li>  ));    **return** <ul>{fruitsList}</ul>;  } |

**방식 2: JSX 내부에서 직접**

|  |
| --- |
| **function** MapRendering2() {  **const** fruits = ["사과", "바나나", "포도"];    **return** (  <ul>  {fruits.map((fruit, idx) => (  <li key={idx}>{fruit}</li>  ))}  </ul>  );  } |

**3. key props의 중요성**

**왜 필요한가?**

1. **아이템 식별**: React가 각 아이템을 정확히 구분
2. **성능 최적화**: 순서 변경 시 필요한 부분만 재렌더링
3. **상태 유지**: 컴포넌트 상태를 올바르게 관리

**올바른 사용법**

|  |
| --- |
| *// ❌ key 없음 - 경고 발생*  {fruits.map(fruit => <li>{fruit}</li>)}  *// ✅ 고유 ID 사용 (권장)*  **const** products = [  { id: 1, name: '사과', price: 2000 },  { id: 2, name: '바나나', price: 1500 }  ];  {products.map(product => (  <li key={product.id}>  {product.name}: {product.price}원  </li>  ))} |

**4. 객체 배열 렌더링 방식**

**방식 1: 컴포넌트 분리**

|  |
| --- |
| **const** users = [  { id: 1, name: "지수", email: "jisu@example.com" },  { id: 2, name: "민준", email: "minjun@example.com" }  ];  **function** User({ name, email }) {  **return** <li>{name} : {email}</li>;  }  **function** UserList() {  **return** (  <ul>  {users.map(user => (  <User key={user.id} name={user.name} email={user.email} />  ))}  </ul>  );  } |

**방식 2: 직접 렌더링**

|  |
| --- |
| **function** UserList() {  **return** (  <ul>  {users.map(user => (  <li key={user.id}>  {user.name}: {user.email}  </li>  ))}  </ul>  );  } |

**5. filter() 함수 활용**

**기본 개념**

* 조건을 만족하는 요소만 새로운 배열로 반환
* **조건부 렌더링**과 **검색 기능**에 활용

**조건부 렌더링 예시**

|  |
| --- |
| **function** TodoList() {  **const** [todos, setTodos] = useState(initialTodos);    *// 미완료 항목만 필터링*  **const** incompleteTodos = todos.filter(todo => !todo.completed);    **return** (  <ul>  {incompleteTodos.map(todo => (  <li key={todo.id}>{todo.text}</li>  ))}  </ul>  );  } |

**검색 기능 구현**

|  |
| --- |
| **function** TodoSearch() {  **const** [todos, setTodos] = useState(initialTodos);  **const** [searchValue, setSearchValue] = useState('');    **const** filteredTodos = todos.filter(todo =>  todo.text.toLowerCase().includes(searchValue.toLowerCase())  );    **return** (  <div>  <input  type="text"  placeholder="검색어를 입력하세요..."  value={searchValue}  onChange={e => setSearchValue(e.target.value)}  />  <ul>  {filteredTodos.map(todo => (  <li key={todo.id}>{todo.text}</li>  ))}  </ul>  </div>  );  } |

**6. CRUD 기능 구현**

|  |
| --- |
| **import** { v4 **as** uuidv4 } **from** 'uuid';  **function** TodoApp() {  **const** [todos, setTodos] = useState([]);  **const** [newTodo, setNewTodo] = useState('');    *// 추가*  **const** handleAddTodo = () => {  **if** (newTodo.trim() === '') **return**;    **const** newTodoItem = {  id: uuidv4(),  text: newTodo,  completed: false  };  setTodos(prevTodos => [...prevTodos, newTodoItem]);  setNewTodo('');  };    *// 삭제 (filter 활용)*  **const** handleRemoveTodo = (id) => {  setTodos(prevTodos => prevTodos.filter(todo => todo.id !== id));  };    **return** (  <div>  <input  value={newTodo}  onChange={e => setNewTodo(e.target.value)}  placeholder="새로운 할일을 입력하세요"  />  <button onClick={handleAddTodo}>추가</button>    <ul>  {todos.map(todo => (  <li key={todo.id}>  {todo.text}  <button onClick={() => handleRemoveTodo(todo.id)}>삭제</button>  </li>  ))}  </ul>  </div>  );  } |