

# 상태 관리와 서버-클라이언트 데이터 공유

Next.js App Router 환경에서는 **서버 컴포넌트**와 **클라이언트 컴포넌트**를 섞어 쓰게 됨.

이 구조에서

상태 관리와 데이터 전달 방식을 이해하는 것이 중요.

## 1. 서버 컴포넌트 → 클라이언트 컴포넌트 데이터 전달

### 서버 컴포넌트 vs 클라이언트 컴포넌트

#### ✅ 서버 컴포넌트(Server Component)

- Next.js 컴포넌트는 기본적으로 서버 컴포넌트
- 브라우저가 아닌 **Node.js** 서버에서 실행됨

➡ 즉, 컴포넌트 렌더링을 브라우저가 아닌 **서버가 먼저 처리**해서 HTML을 만든 다음, 이것을 브라우저에게 전달

#### 1. 서버 컴포넌트에서 데이터를 fetch

```
// Server Component
// app/page.tsx

async function Page() {
  const data = await getData(); // 서버에서 데이터 fetch
  return <Client data={data} />; // 데이터를 클라이언트 컴포넌트에 넘김
}
```

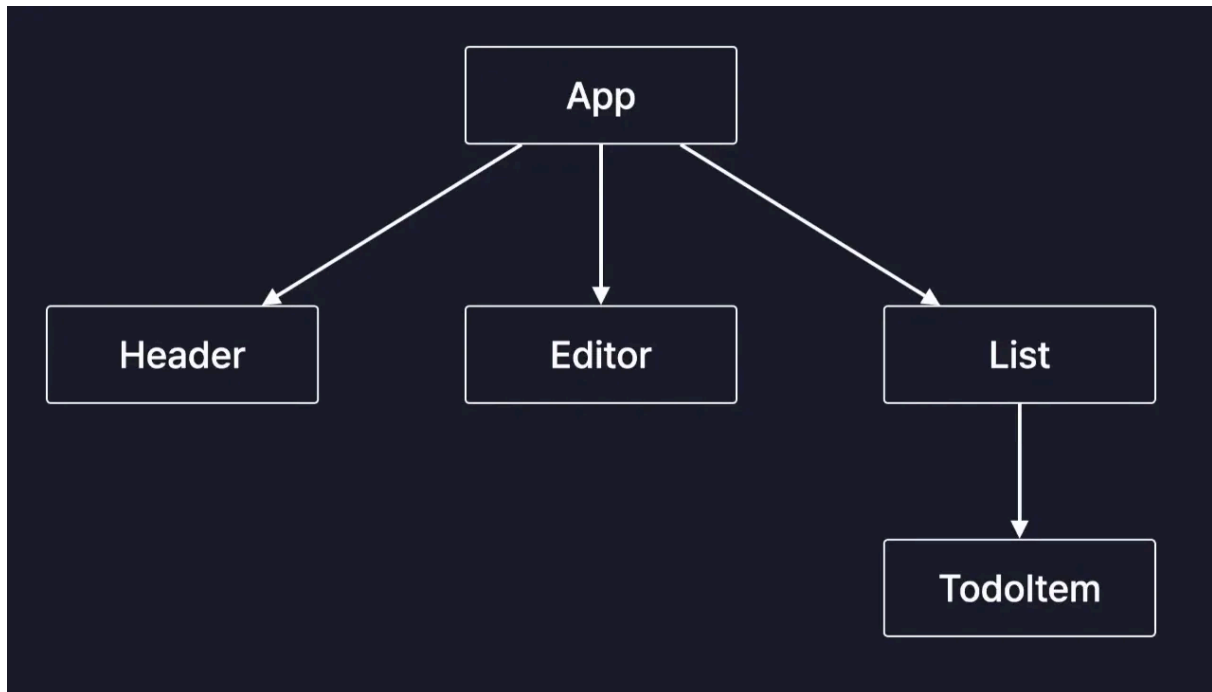
#### 2. 클라이언트 컴포넌트는 props로 받아서 상태화 및 UI 상호작용 처리

```
// 'use client'
export default function Client({ data }) {
```

```
const [info, setInfo] = useState(data); // 클라이언트에서 상태로 전환
...
}
```

- `'use client'` : 해당 컴포넌트가 **React 클라이언트 컴포넌트**라는 것을 Next.js에게 알리는 선언

## 2. Context API 활용하기



- App 컴포넌트에 있는 함수들을 TodoItem 컴포넌트에 전달해야 하는 상황
- 그러려면 중간에 **List 컴포넌트**를 거쳐야 했음.

### Context API란?

#### | 여러 컴포넌트에서 공유할 수 있는 전역 상태를 만들기 위한 React의 내장 기능

페이지 전체에서 로그인 상태, 테마, 다크모드처럼 **공통된 상태**를 써야 할 때, Context를 쓰면 중간에 props를 계속 넘기지 않아도 됨.

## Context 활용 예제

```
// App.jsx
import { createContext } from 'react';

const TodoContext = createContext();

// App 컴포넌트 내부
function App(){
  ...
  return (
    <>
      <div className='App'>
        <Header />

        <TodoContext.Provider
          value={{
            todos,
            onCreate,
            onUpdate,
            onDelete
          }}
        >
          <Editor />
          <List />
        </TodoContext.Provider>

      </div>
    </>
  )
}
```

## 3. 외부 상태 관리 라이브러리 (Zustand, Jotai 등)

외부 상태 관리 라이브러리란 ?

| Context API보다 더 유연하고 확장성 높은 전역 상태 관리 도구

## 왜 Context API 대신 외부 라이브러리를 쓸까?

문제점 (Context API)	해결 방법 (외부 라이브러리)
상태가 많아질수록 코드가 복잡해짐	훨씬 간결한 API
상태 변경이 느릴 수 있음 (re-render 비용 큼)	성능 최적화 구조 내장
상태 파일이 여러 개 필요	단일 스토어로 구성 가능

## ✅ Zustand 기본 예시

### 1. 스토어 정의

앱 전체에서 공유할 수 있는 '전역 상태 저장소'를 만든다는 뜻  
쉽게 말해,  
`useState()` 로 만든 상태를 하나의 파일에 묶어서 보관하는 거라고 보면 됨.

```
// stores/useCounterStore.ts
import { create } from 'zustand';

// useCounterStore라는 이름의 스토어 정의
export const useCounterStore = create((set) => ({
  count: 0,
  increase: () => set((state) => ({ count: state.count + 1 })),
}));
```

### 2. 컴포넌트에서 사용

```
'use client';
import { useCounterStore } from '@stores/useCounterStore';

export default function Counter() {
  const { count, increase } = useCounterStore(); // Zustand 스토어에서 꺼내 씬
  return <button onClick={increase}>Count: {count}</button>;
}
```

## ✅ Jotai 기본 예시

## 1. Atom 생성

```
// atoms/counterAtom.ts
import { atom } from 'jotai';

export const counterAtom = atom(0); // 초기값 0
```

- `counterAtom` 이라는 상태 변수 정의
- `counterAtom` 은 리액트의 전역 상태처럼 앱 어디서든 `useAtom()` 을 통해 읽고, 수정 가능

## 2. 컴포넌트에서 사용

```
'use client';

import { useAtom } from 'jotai';
import { counterAtom } from '@atoms/counterAtom';

export default function Counter() {
  const [count, setCount] = useAtom(counterAtom); // 상태를 읽고 쓰기
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

## 결론 !

- **Zustand**는 여러 상태를 한 번에 다루기 좋은 '스토어 중심' 라이브러리
  - 복잡한 액션 중심 앱
  - 다중 상태 관리
  - 구조화된 앱에 적절
- **Jotai**는 각각의 상태를 독립적인 단위로 다루는 '선언형 원자 상태 관리' 방식
  - 간단하고 기능 중심 앱에 적절



### 비유로 이해하기 (by ChatGPT)

Zustand는 커다란 통 하나에 모든 상태를 담고,

Jotai는 작은 병 여러 개에 각각 나눠 담는다고 보면 됨

## 4. 서버와 클라이언트 간의 상태 동기화

### Hydration Mismatch 란 ?

: 서버에서 가져온 데이터와 클라이언트가 렌더링하면서 사용하는 데이터가 서로 어긋나는 현상

### ✓ 해결 전략 3가지

전략	설명	라이브러리	추천 상황
props로 초기화 + useState/useEffect	서버에서 넘긴 값을 초기 상태로 쓰되, 클라이언트에서 업데이트할 수 있게 함	기본 React	초기 데이터만 있으면 되는 경우
CSR + SWR/React Query	클라이언트에서 데이터를 fetch하고 캐싱으로 관리	SWR, React Query	데이터가 자주 변하거나 최신화가 중요할 때
서버 컴포넌트 전용 처리	서버에서만 렌더링해서 상태 변화가 없도록 함 (읽기 전용)	Next.js App Router	보안/SEO 중요, 상호작용 불필요한 경우

### CSR + SWR / React Query 전략

사용자 브라우저에서 직접 데이터를 가져오고,  
한 번 가져온 데이터는  
다시 서버에 요청하지 않고 저장해두는 방식

- **SWR ( Stale While Revalidate )**  
→ 오래된 데이터를 먼저 보여주고, 백그라운드에서 최신 데이터로 업데이트
- **React Query:** 데이터 fetch + 캐시 + 로딩/에러 관리 자동 처리