1.1 초기 CSR 과 초기 SSR

(1) CSR

초기 CSR 은 주로 기본적인 HTML, CSS, 그리고 JavaScript 로 구현되었으며,

AJAX 를 사용하여 데이터를 비동기적으로 로드하는 방식이 일반적.

초기에는 브라우저 기술이 제한적이었기 때문에 복잡한 사용자 인터페이스를 구현하기 어렵고, 성능 문제나 호환성 문제가 발생 가능성이 있었으며,

jQuery 와 같은 라이브러리를 사용하여 DOM 조작과 AJAX 요청을 쉽게 처리하는 방식이 주류였습니다.

(2) SSR

초기 SSR 은 PHP, ASP.NET 과 같은 서버 사이드 언어와 프레임워크를 같이 사용하여 렌더링되었으며, 서버가 요청을 수신하여 HTML 을 생성하고 이를 클라이언트에 전달하는 방식.

사용자가 요청한 페이지를 서버가 처리하여 전체 HTML 을 생성한 후 클라이언트에 전달하였으며,

이 과정에서 HTML 구조를 동적으로 생성하기 위해 서버 사이드 템플릿 엔진(예: EJS, JSP 등)을 사용했습니다.

초기에는 페이지 로드 시 모든 리소스가 서버에서 처리되고 전송되므로, 페이지 전환이 느리고 서버 부하가 크며, 크롤러가 JavaScript 를 처리하지 못하는 경우 SEO 에 불리한 점이 있었습니다.

1.2 현 CSR과 현 SSR

(1) CSR

현재는 React, Vue, Angular와 같은 현대적인 JavaScript 프레임워크가 널리 사용되고 있으며,

이러한 프레임워크는 컴포넌트 기반 아키텍처를 제공하여 유지보수성과 재사용성을 극대화

상태 관리 도구인 Redux, Vuex 등을 통해 복잡한 애플리케이션의 상태 관리를 용이하게 하고, 라우팅을 통해 페이지 전환을 부드럽게 처리하며,

브라우저의 성능이 향상되고, 자바스크립트 엔진이 발전하면서 CSR 방식의 렌더링 시간도 단축됨

(2) SSR

현재는 Node.js와 Express.js 같은 현대적인 서버 사이드 JavaScript 환경, 그리고 Next.js, Nuxt.js와 같은 프레임워크가 SSR을 지원하며,

이를 통해 같은 코드베이스에서 클라이언트 렌더링과 서버 렌더링을 혼합하여 사용가능합니다.

API를 통해 데이터를 비동기적으로 가져오며, 서버가 요청을 처리하는 동안 필요한 데이터만 패칭하고,

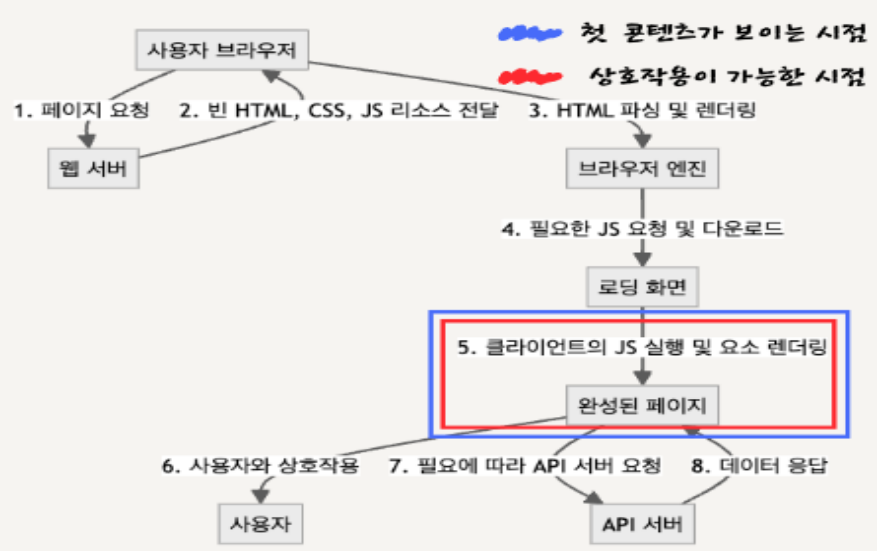
페이지가 로드되기 전에 HTML을 생성하여 클라이언트에게 빠르게 전송할 수 있도록 최적화한 후, 서버 측에서 미리 렌더링된 페이지는 사용자가 브라우저에서 빠르게 보기 시작 가능.

최근의 SSR 프레임워크는 SEO를 고려하여 메타 태그와 같은 HTML 메타데이터도 서버에서 동적으로 처리할 수 있게 설계되어 있으며,

현재의 SSR에서는 클라이언트와 서버 간의 인터랙션을 더욱 효율적으로 처리할 수 있으며, 클라이언트 상태 관리와 데이터 패칭에 대한 통합된 접근 방식을 제공.

1.2.1 각 구동방식

(1) CSR



① 사용자의 브라우저 방문, 서버에게 HTML 요청

② 서버는 빈 HTML, CSS, JS 리소스를 전달

③ 클라이언트는 HTML 을 파싱하고 렌더링

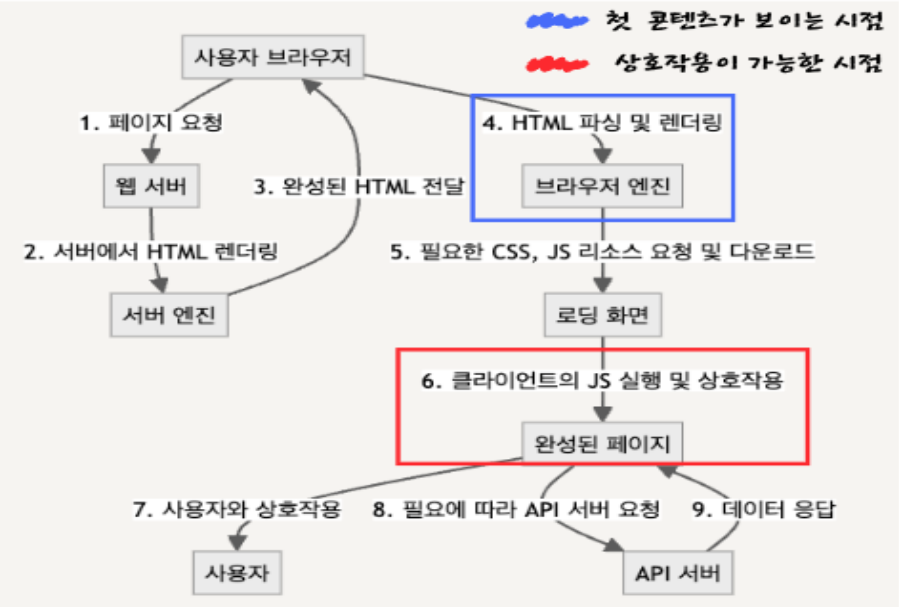
- 이때까지, 빈 HTML 화면이 표현됨

④ HTML 을 파싱하는 과정에서 "script" 태그를 만나게 되면 해당 JS 파일을 요청

⑤ 브라우저는 JS 를 실행하고 화면에 콘텐츠를 렌더링

- 이때 빈화면에서 실제 UI 가 화면상에 표현됨. (이때부터 사용자와의 상호작용이 가능)

(2) SSR



①사용자의 브라우저 방문, 서버에게 HTML 요청

② 서버는 렌더링 준비가 된 HTML 을 전달

③ 클라이언트는 HTML 을 파싱하고 렌더링

- 이 시점에서 화면상에 콘텐츠가 표현됨 (서버에서 생성해줌)

④ HTML 을 파싱하는 과정에서 "script" 태그를 만나게 되면 해당 JS 파일을 요청

- bundle.js 파일을 다운로드 받는 시점

⑤ 브라우저가 JS 를 실행하고 구성요소를 하이드레이션

- 이때부터 사용자와의 상호작용이 가능

▶ 즉 사용자에게 빠른 초기 로딩과 JS 를 실행하고나서 원활한 상호작용을 제공

메모 포함[강1]: 서버에서 렌더링된 정적 HTML 에 클라이언트 사이드의 동적인 기능을 연결하는 과정

1.3 리액트 CSR 과 SSR

(1) 리액트 CSR

- index.html - 빈 div 태그
- index.js - 빈 div 태그에 App.js 를 동적 삽입

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <App />
);
```

- 렌더링 후 화면소스에서 App.js 의 소스코드는 볼 수 없음 → SEO 불리

```
29    <body>
30      <noscript>You need to enable JavaScript to run this app.</noscript>
31      <div id="root"></div>
32    </body>
```

(2) 리액트 SSR

- 서버 코드 - renderToString()를 통하여 App.js 를 문자열로 반환하여 응답

```
app.get("*", (req,res) => {
    const renderString = renderToString(<App/>);
    const result = html
    .replace(
        '<div id="root"></div>',
        `<div id="root">${renderString}</div>`
    ).replace('__DATA_FROM_SERVER__', JSON.stringify(initialData))

    res.send(result);
});
```

- index.js - render() → hydrate() 변경 (정적 요소에 이벤트 추가)
  - ReactDom.hydrate(<App />, document.getElementById('root'));
- 렌더링 후 화면소스에서 App.js 의 소스코드를 볼 수 있음 → SEO 유리

```
<body>
    <div id="__next">
        <div id="root">
            <h1>NextJS App</h1>
            <button>count is
            <!-- -->
            0</button>
            <p>Some random content here</p>
        </div>
```

1.4 CSR + SSR 혼합 시(하이브리드 렌더링)

1.4.1. 구동방식

① Next.js 는 서버에서 컴포넌트를 읽어 HTML 을 생성하고, 이를 클라이언트에 응답한다.

② 클라이언트는 말그대로, 정적인 HTML 을 전달받은거다. 여기에 React 로 작성한 자바스크립트 코드와 연결을 해줘야하는데 번들링된 JS 도 같이 내려받는다.

③ 클라이언트는 전달받은 HTML 을 hydrateRoot() 를 호출하여 렌더링된 HTML 에 자바스크립트 코드를 연결시켜준다(React 로 채워주기)

1.4.2. 혼합 시 장점

1. 명확한 역할 분담: SSR 에서는 백엔드가 HTML 을 렌더링하고, 프론트엔드는 그 HTML 을 사용자와 상호작용하는 데 집중한다. 이로 인해 역할이 명확히 나뉘어 협업이 원활해진다.

2. API 와 페이지 구조의 명확화: 백엔드는 데이터와 페이지 구조를 정의하고, 프론트엔드는 이를 기반으로 사용자 인터페이스를 설계한다.

▶ 이로 인해 API 와 페이지 구조에 대한 명확한 기준을 세울 수 있어 협업이 용이하다.

1.5 각 장단점

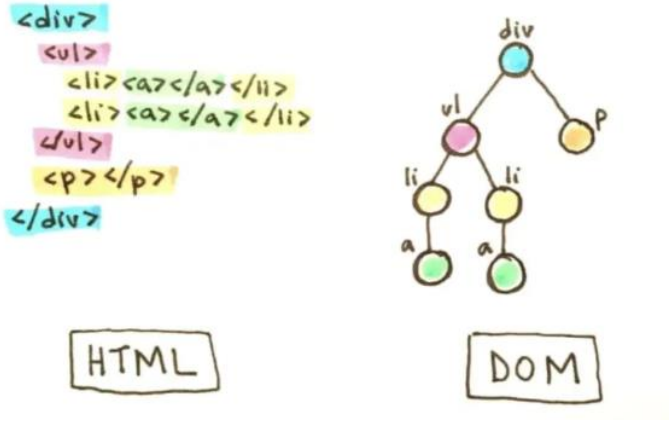
	CSR	SSR
첫 페이지 로딩	HTML, CSS, 모든 스크립트를 한 번에 불러와서 첫 페이지 로딩 시간 느림	필요한 부분의 HTML 과 스크립트만 불러오게 돼서 첫 페이지 로딩 시간 빠름
다른 페이지로의 이동	이미 첫 페이지 로딩할 때 나머지 부분을 구성하는 코드를 받아왔기 때문에 빠름	첫 페이지를 로딩한 과정을 정확하게 다시 실행한다. 그래서 더 느림.
SEO 에 대한 적합성	자바스크립트를 실행시켜 동적으로 컨텐츠가 생성되기 때문에 SEO 에 부적합	서버 사이드에서 컴파일 되어 클라이언트로 넘어오기 때문에 크롤러에 대응하기 용이
서버 자원 사용	클라이언트에 일감을 몰아주기 때문에 서버의 부하 적음	매번 서버에 요청하기 때문에 서버 자원 사용 ↑
캐싱	가능	불가능
보안성	- 쿠키, 로컬스토리지에 저장되는 사용자 정보 도용에 취약 - 기능이 클라리언트에서 수행되므로 로직의 노출가능성이 있음	- 서버에서 렌더링 시 escaping 과 같은 조치를 취할 수 있어 안정적 - XSS 공격에 대한 위험이 낮음
사용자 경험	부드러운 페이지 전환	화면 깜빡임 가능성
개발 난이도	높음(복잡한 SPA 구현)	낮음(일반적인 웹 개발)
대표 기술 및 사이트	- React, Vue - facebook 등 각종 SNS	- Next.js, node.js - 네이버 블로그 및 신문사 등의 사이트

메모 포함[강2]: 검색 엔진 최적화로 웹사이트가 검색 결과에 더 잘 보이도록 최적화하는 과정

2.1 Virtual DOM / Real DOM

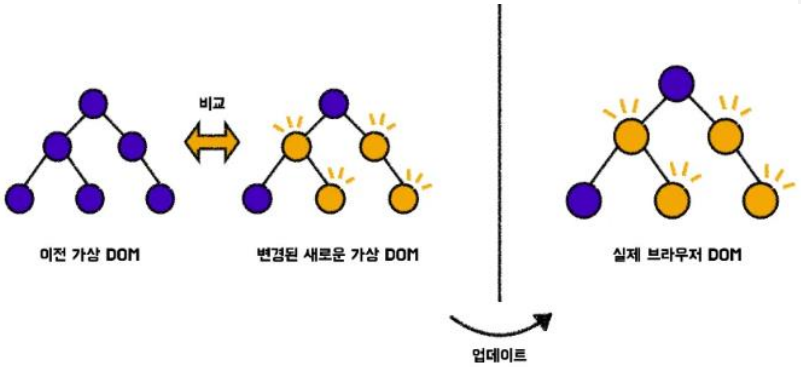
(1) Real DOM

- 브라우저에 실제로 렌더링되는 HTML 문서 객체 모델
- 변경사항이 생기는 즉시 웹에 반영됨
- CSSOM, 레이아웃, 이벤트 핸들러와 같은 내부 속성 등 포함



(2) Virtual DOM

- JavaScript 객체로 관리되는 Real DOM 의 가벼운 복사본
- Real DOM 보다 메모리를 적게 사용 (Real DOM 보다 데이터 구조가 간단)
- 필요하지 않은 브라우저 엔진 관련 데이터(스타일, 레이아웃 등)는 포함하지 않음



(3) React Virtual DOM

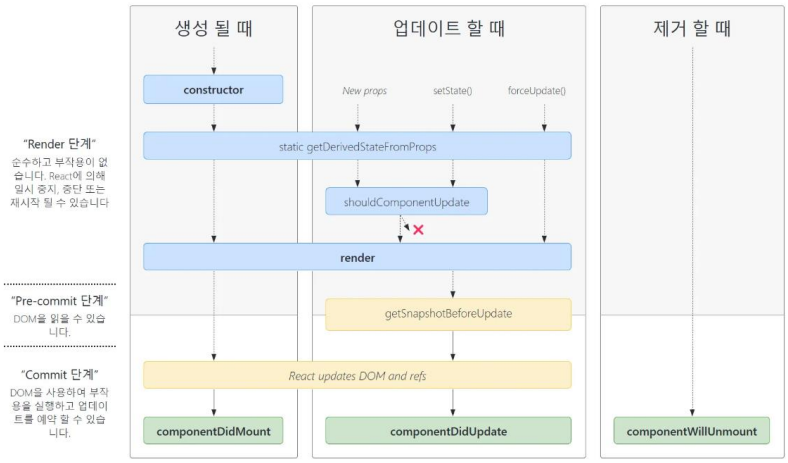
- 2 개의 Virtual DOM 사용
  - 현재 Virtual DOM 과 업데이트를 적용한 Virtual DOM 을 비교(Diffing)하기 위함
  - 두 DOM 간의 차이를 계산(Patching)하고, 필요한 변경사항만 Real DOM 에 반영(Batch Update)
  - 변동사항이 없을 때도 두 Virtual DOM 은 메모리에 유지되며, 이는 성능 최적화를 위해 되어있는 설계

(4) Virtual DOM vs. Real DOM

- Virtual DOM 과 Real DOM 이 브라우저에 각각 떠 있을 때 메모리 사용량 차이
  - Virtual DOM - 적음
  - Real DOM - 많음
- React(Virtual DOM 2 개 + Real DOM 1 개)와 jQuery(Real DOM 1 개)가 각각 브라우저에 떠 있을 때 메모리 사용량 차이
  - React(Virtual DOM 2 개 + Real DOM 1 개) - 많음
  - jQuery(Real DOM 1 개) - 적음
- React 가 Virtual DOM 2 개 + Real DOM 1 개를 사용하는 이유
  - 추가적인 메모리 소비를 감수하고도 상태 변화가 많거나 렌더링이 자주 발생하는 상황에서 Real DOM 만 사용하는 것보다 효율적이기 때문

2.2 React Life Cycle

- 라이프사이클은 컴포넌트가 생성되고, 업데이트되며, 제거되는 과정에서 호출되는 일련의 단계와 메서드를 말함
- React 에서는 이 과정을 통해 개발자가 특정 시점에 원하는 동작을 수행할 수 있도록 제공하며, 컴포넌트의 상태 관리, 외부 API 호출, DOM 조작, 메모리 정리 등을 위해 사용됨



(1) Mount 생성

- 컴포넌트가 DOM 에 삽입되는 단계로, 초기 렌더링을 준비하고 실행
- 메서드 : constructor(), render(), componentDidMount()

(2) Update 업데이트

- 상태(state)나 속성(props)이 변경되어 컴포넌트가 다시 렌더링되는 단계로, 필요한 부분만 DOM 을 갱신
- 메서드 : shouldComponentUpdate(), render(), componentDidUpdate()

(3) Unmount 제거

- 컴포넌트가 DOM 에서 제거되는 단계로, 정리 작업을 수행
- 메서드 : componentWillUnmount()

2.3 React 를 써야하는 이유

(1) 유지보수 인력

- React 에 대한 요즘 개발자의 선호도
- 앞으로 jQuery 보다 React 를 능숙하게 다루는 인력이 더 많아질 것, 새로운 인력에게 기대할 수 있는 기술
- 한 시스템을 오래 운영한다고 가정했을 때, jQuery 를 사용하는 사람은 점점 줄어들 것
- 앞으로 유지보수를 맡게 될 '자라다'도 사용자와의 상호작용이 있는 애플리케이션으로 React 로 구현되어 있음

(2) 범용성

- 멀티플랫폼 환경(냉장고 내 디스플레이 등)에서도 SPA 방식 지향
- jquery 보다 리액트로 SPA 를 구현하기 더 쉬움