**1. 서버사이드 렌더링이란 무엇인가**

**- Rendering** : 어떠한 웹 페이지 접속시, 그 페이지를 화면에 그려주는 것. 요청시마다 서버에서 처리 한 후 HTML&View를 생성하여 새로고침으로 페이지에 대한 응답(View)(새로고침이 일어나며 서버에 새로운 페이지에 대한 요청을 하는 방식)

1) 로더가(Loader)가 서버로부터 정보들을 불러옴  
2) 파싱(Phasing)을 통해 문서를 DOM 트리 만듦  
3) DOM트리가 구축되는 동안 브라우저는 렌더 트리를 구축   
4) css 설정/레이아웃 위치 지정  
5) 렌더링 트리가 그려짐

- 전통적인 웹 어플리케이션은 서버사이드렌더링 방식을 사용(전통 웹 방식 != SSR)즉, 브라우저에 나타나는 형태 그대로를 HTML로 만들어 제공하고, 브라우저는 HTML을 표시하는 방식이었다. 이런 방식을 사용하다가 일부 HTML과 Script만 브라우저로 전달하고, 브라우저에서 Script를 실행시켜 서버에서 데이터를 조회하여 HTML을 생성하는 방식을 사용하게 되었다.

- 웹에서 많은 정보와 기능이 많아지면서 싱글페이지어플리케이션(SPA)개념이 생김

**서버사이드렌더링(Server-Side-Rendering)의 장점**

- SSR의 경우 view를 서버에서 렌더링하여 가져오므로 첫 로딩이 매우 짧음(view를 보기까지)

- 물론 js파일을 모두 다운로드하고 적용하기 전까지는 그 어떤 인터렉션에도 반응하지는 않지만, 사용자 입장에서는 로딩이 매우 빠르다고 느껴짐.

- SEO(Search Engine Optimization) 검색엔진 최적화에 유리함 (js 라이브러리/프레임워크를 사용하면서 고질적인 문제는 seo이다. 그냥 미리 그려지지 않고 빈 껍데기만 제공된다면 검색엔진이 당연하게도 내용을 검색하는데 어려움이 있음 but, SSR은 서버에서부터 페이지를 제공할 때 컨텐츠가 담겨있기 때문에 SEO에 대한 걱정을 덜 수 있음)

**서버사이드렌더링(Server-Side-Rendering)의 단점**

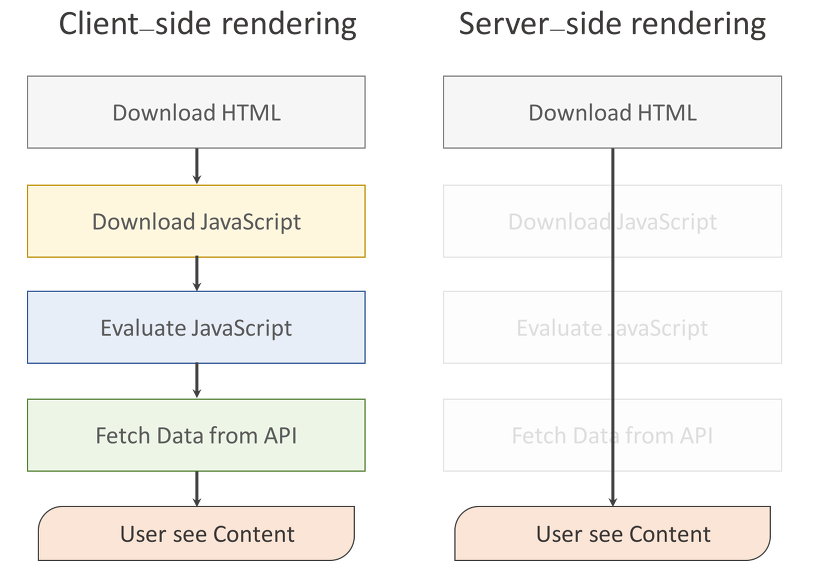
- 장점을 제공하는 만큼 사용하는데에 있어서 어려움과 번거로움 (서버사이드 렌더링을 구현하게 된다면, 프로젝트의 구조가 많이 복잡해지게 됩니다.

- 단순히 렌더링만 하는것은 그렇게 큰 문제는 아니지만, 리액트 라우터, 리덕스 등의 라이브러리와 함께 연동해서 사용하면서, 서버에서 데이터를 가져와서 렌더링을 해줘야하는 경우엔 조금 어려워질수도 있습니다.)

- view 변경 시 서버에 계속 요청을 해야하므로 서버에 부담이 큼, 정보가 많은 B2C 웹 서비스 등에는 서버 부담이 큼(요청 시 마다 새로고침이 일어나며 페이지를 로딩할 때마다 서버로부터 리소스를 전달받아 해석하고 화면에 렌더링 하는 방식이기 때문이다. 예를들어 현재 주소에서 동일한 주소를 가리키는(갖고있는) 버튼을 눌렀을때, 설정페이지에서 필요한 데이터를 다시 가져올 수 없다. 이것은 사용자와 인터랙션이 많은 요즘 웹앱에게 충분하지 않는 방법일 수 있다. 렌더링을 서버쪽에서 하는 것은 그 만큼 렌더링을 위한 서버 자원이 사용되는 것이고, 불필요한 트래픽도 낭비되는 것이다.

- 서버사이드 렌더링을 구현하게 된다면, 프로젝트의 구조가 많이 복잡해지게 됩니다. 단순히 렌더링만 하는것은 그렇게 큰 문제는 아니지만, 리액트 라우터, 리덕스 등의 라이브러리와 함께 연동해서 사용하면서, 서버에서 데이터를 가져와서 렌더링을 해줘야하는 경우엔 조금 어려워질수도 있습니다. (대안) 동일한 페이지는 특정기간 동안 캐싱하여 성능을 최적화 할 수 있음

**2. SPA란 무엇이고 어떤 장점이 있는가.**



Client Side Rendering :클라이언트에서 자바스크립트를 통해 렌더링 하는 방식

Single Page Application (SPA) 는 CSR 방식.

SPA는 브라우저에 로드되고 난 뒤에 페이지 전체를 서버에 요청하는 것이 아니라 최초 한번 전체 페이지를 다 불러오고 HTML에 번들링 된 js가 실행되면서 페이지 렌더링을 한다. 그리고 렌더링 후 클라이언트 요청에 따라 응답 데이터만 다시 페이지의 특정 부분만 렌더링하는 웹 어플리케이션 개념이다.

**클라이언트사이드렌더링의 장점**

**트래픽감소**필요한, 변경된 데이터만 받아서 그림

**사용자경험**새로고침이 발생하지 않아 사용자가 네이티브 앱과 비슷한 경험을 할 수 있다.

**클라이언트사이드렌더링의 단점**

**검색엔진**자바스크립트 위주로 돌아가는 프로젝트는 자바스크립트 엔진이 돌아가지 않으면 원하는 정보를 표시해주지 못함. 크롬에서 react로 만든 웹앱의 소스를 확인하면, 내용이 비어있다. 그렇기때문에 검색엔진 크롤러가 데이터들을 제대로 수집하지 못한다. 하지만 짱짱맨 구글의 검색엔진은 자바스크립트 엔진이 내장되어있다. 하지만 네이버, 다음 등의 검색엔진은 제대로 크롤링하지 못하기때문에 서버사이드렌더링을 따로 구현해야한다.

**SPA에서 서버사이드렌더링을 적용시켰을때 단점**

**프로젝트의 복잡도**React에서 서버사이드렌더링을 구현할 경우 Router와 Redux와 함께 사용하다보면 복잡해질 수 있다.

**성능의 악화 가능성**서버사이드 렌더링을 하게 될 때는, ReactDOMServer.renderToString 함수를 사용한다. 이 함수는 동기적으로 작동한다. 그래서, 렌더링하는동안은 이벤트루프가 막히게 된다. 하지만.. ! 라이브러리를 통하여 비동기식으로 작동하게끔 코드를 작성 할 수 있다.

**대안**

**메타태그만 넣어주기**서버쪽에서 라우트에 따라 필요한 메타태그만 넣어주는 것. 그러면, 크롤러에선 해당 페이지에 대한 기본 정보는 얻어 갈 수 있게 된다.

**Prerender**이 서비스는 오직 검색엔진 최적화를 위해서만 사용

출처 : <https://velog.io/@zansol/%ED%99%95%EC%9D%B8%ED%95%98%EA%B8%B0-%EC%84%9C%EB%B2%84%EC%82%AC%EC%9D%B4%EB%93%9C%EB%A0%8C%EB%8D%94%EB%A7%81SSR-%ED%81%B4%EB%9D%BC%EC%9D%B4%EC%96%B8%ED%8A%B8%EC%82%AC%EC%9D%B4%EB%93%9C%EB%A0%8C%EB%8D%94%EB%A7%81CSR>

**3. 리액트에 사용되는 디자인패턴은 어떤 것들이 있는가. 각각 설명하시오.**  
React Hooks의 등장으로 보다 선언적인 프로그래밍 스타일을 추구할 수 있고 함수형 컴포넌트를 적극적으로 사용할 수 있게 됐다.하지만 앱의 규모가 커지면, 여전히 코드는 이해하고 관리하기가 어려워진다. 컴포넌트에 동기화 된 데이터와 중복된 코드들로 인해 훌륭하게 설계된 컴포넌트들이 빠르게 망가지기 시작한다. 이러한 상황에서 우린 디자인 패턴을 적용해봄으로써 앱의 아키텍처를 향상시키고 관련없는 로직으로 인해 컴포넌트가 커지는 것을 막을 수 있다.

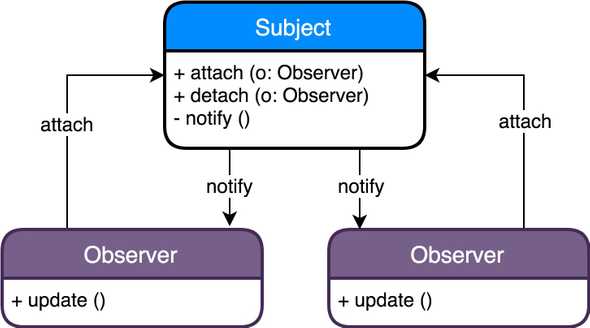
**디자인 패턴**

디자인 패턴은 다양한 상황에서 코드에 적용할 수 있는 잘 정의된 솔루션이다. 디자인 패턴은 모든 언어에 적용된다. 즉 작성해야 할 코드가 아니라 어떤 클래스/객체를 사용할 지와 서로가 영향을 어떻게 주는 지에 대한 일종의 개요 같은 것이다.

디자인 패턴은 그 자체로 개발자들끼리 소통하는데에 필요한 강력한 기반을 제공한다. 코드에서 익숙한 디자인 패턴을 사용하면 그 코드는 자체로 소통할 수 있는 문서가 된다. 코드에서 디자인 패턴 용어나 인터페이스를 인지했을 때, 써드 파티 혹은 동료의 코드를 이해하는 것이 더 쉬워진다.

출처 : <https://delivan.dev/react/programming-patterns-with-react-hooks-kr/>

**Observer 패턴**



서로 연결되진 않았지만 데이터가 동기화 되어 있으면 하는 두 컴포넌트가 존재할 때 유용하다. 동작 방식은 먼저 Observer라고 불리는 것이 자신을 Subject에 attach 한 후 계속 기다린다. 그리고 Subject가 변경되면 자신의 최신 데이터를 attach 된 Observer 들을 한테 notify 한다.

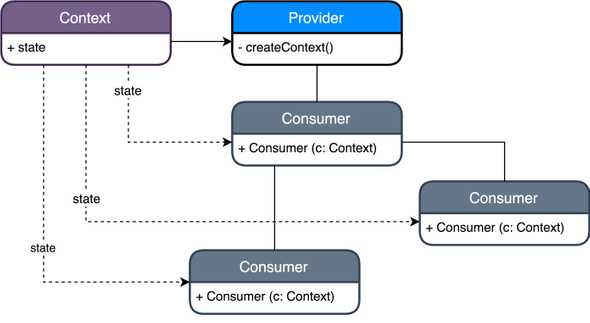
**장점**

컴포넌트 간의 낮은 의존성: 컴포넌트들이 서로 동기화 되기 위해 같은 트리에 있을 필요가 없다.  
단방향 데이터 흐름: 변화가 한 곳에서만 일어나며 추적하기 쉽다.

**단점**

메모리 누수: 사용하지 않는 Observer들을 피하기 위해 사용되지 않는 컴포넌트를 Subject로 부터 detach할 필요가 있다.

**Provider 패턴**



Provider 패턴은 .Net Microsoft 앱 내에서 클래스의 초기화에 사용되는 다양한 조미료(React에선 props)를 제공하기 위해 처음 사용되었다. 이 패턴은 컴포넌트들이 ‘전역’ 상태를 가질 수 있게 하는 React Context API에 유용하게 사용되었다.

당신이 다수의 컴포넌트에서 사용가능한 일반적인 객체를 만들기를 원하고 그 객체가 변할때 마다 자식 컴포넌트들이 업데이트 되길 원할 때 유용하다.

이 패턴의 주요한 이점은 트리 내의 컴포넌트마다 각각 다 props로 값을 넘겨야 하는 상황을 피할 수 있다는 것이다. 이 문제는 보통 “Prop Drilling”으로 잘 알려져 있다.

동작 방식은 먼저 Provider가 컴포넌트 트리의 상위 레벨에 설정된 Context 객체에 일부 값들을 설정한다. Consumer라고 불리는 자식 컴포넌트들은 props로 값들을 전달받는 것 대신, context로부터 값들을 직접 가져올 수 있다.

**장점**

prop drilling을 예방: 부모와 자식 컴포넌트간의 의존성을 덜어준다.  
단방향 데이터 흐름

**단점**

보이지 않는 복잡성: 어떤 컴포넌트가 부모이고 데이터가 어디서 오는지 알기 어려움  
전역적인 값을 사용하게 함 => 전반적으로 앱의 응집력이 높아짐 (React 공식 문서에는 Context를 적게 사용하는 것을 추천한다)

주의사항

출처 : <https://delivan.dev/react/programming-patterns-with-react-hooks-kr/>