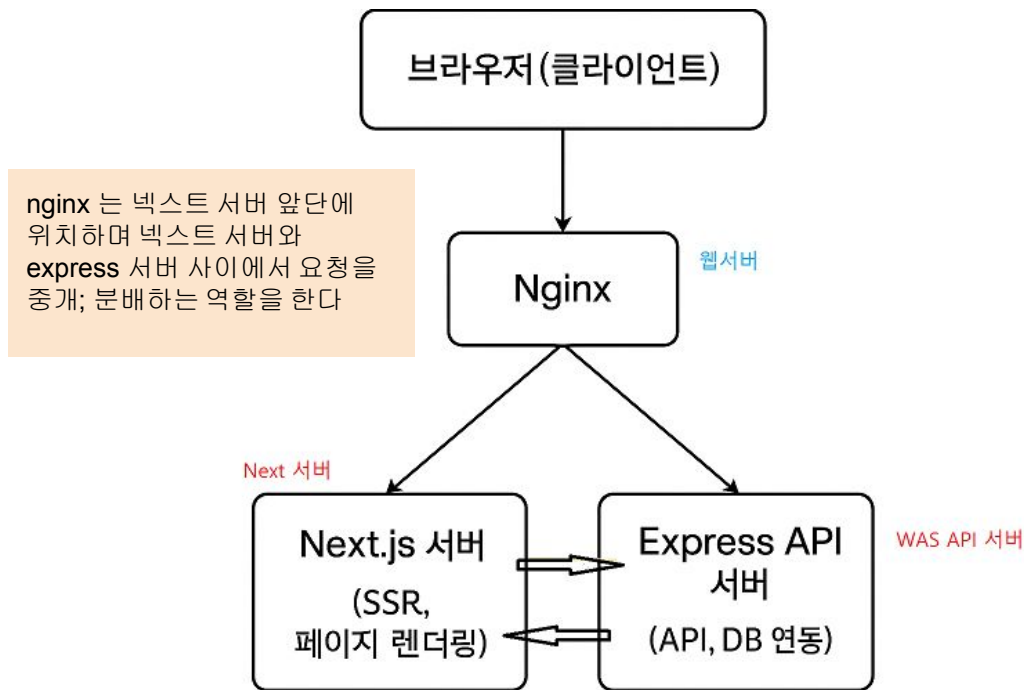

Next.js

백성애 2025-05-12

웹서비스 구조도:

Nginx 기반 Next.js(SSR) + Express(API) 연동 아키텍처



- 브라우저(클라이언트)가 요청을 보냅니다.
- 요청은 Nginx (리버스 프록시, 로드밸런서, SSL 처리)에 도착합니다.
- Nginx는 요청 경로에 따라
 - Next.js 서버(포트 3000)로 전달하거나
 - Express API 서버(포트 8080)로 전달합니다.
- Next.js 서버는 SSR 및 페이지 렌더링을 담당합니다.
- Express 서버는 API 처리와 DB 연동을 담당합니다.

Next.js 란?

React기반 웹 개발 프레임워크. (Vercel사에서 개발)

UI는 React 로 구성하며,

Next.js 는 페이지 라우팅, SEO(Search Engine Optimization) 를 위한 서버 사이드 렌더링(SSR), 정적 사이트 생성(SSG), API 라우팅 등을 지원한다

또한 빌트인 성능 최적화, 증분 정적 콘텐츠 생성(ISR) 등을 제공하여 동적이고 빠른 React 애플리케이션을 구축하도록 도와주는 오픈소스 프레임워크

Next.js의 주요 특징

**Next.js = 빠르고 유연한
React 웹 프레임워크**

- SSR + SSG + CSR
조합 가능
- 백엔드 기능 내장
(API Routes)

기능	설명
SSR	서버에서 HTML을 렌더링해 SEO에 유리
SSG	빌드 시 정적 HTML 생성 → 빠름
CSR	React처럼 동작하는 클라이언트 렌더링도 가능
API Routes	백엔드 없이 API 엔드포인트 생성 가능
이미지 최적화	next/image 컴포넌트로 자동 최적화
자동 라우팅	pages 폴더 구조 기반 라우팅
TypeScript 지원	내장 TS 설정으로 타입 안전성 제공

백트의 경우 처음에 index.html 을
드한 뒤 js 번들을 실행시켜 UI
포넌트를 렌더링 함

색엔진 크롤러는 js는 실행하지
거나 느려서 html을 인덱싱 하게
는데 실제 내용이 있는 html이
니라 빈 화면을 인덱싱 하게 됨

라서 검색 엔진에 콘텐츠 노출이
한적일 수 밖에 없게 됨

Next.js의 Pre-rendering 특징

Pre-rendering을 이해하기 위해 먼저 React의 CSR 방식을
살펴보자

No Pre-rendering (Plain React.js app)

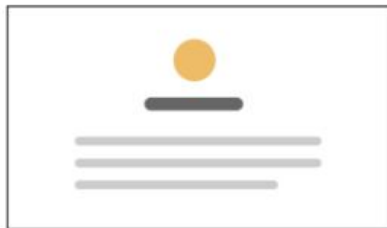
Initial Load:

App is not rendered



JS loads
→

Hydration: React components are
initialized and App becomes interactive



기본적으로 React는 CSR(client Side
Rendering)을 한다.

이는 왼쪽 그림처럼 웹사이트를 요청했을
때 빈 html(index.html)을 가져와 script를
로딩하기 때문에, 첫 로딩 시간도
오래걸리고 Search Engine
Optimization(SEO)에 취약하다는 단점이
있다

React 앱- CSR방식

- 초기 HTML에는 `<div id="root"></div>`만 있음
- 브라우저가 JS 파일을 다운로드하고 실행한 후에야 UI가 나타남

단점

- 초기 로딩 시간이 느림
- **FCP**가 늦어짐 (처음 글자/이미지 보이기까지 시간)

CSR 방식

애플리케이션에 필요한 모든 정적 리소스를 최초 단 한 번만 다운로드하여 렌더링하고, 이후 새로운 페이지에 대한 요청이 있을 때에는 페이지 갱신에 필요한 데이터만을 내려받아 리렌더링 한다

장점: 페이지 이동이 빠르다

단점: **FCP(First Contentful Paint)**가 늦어짐 => 초기 접속 속도가 느리다

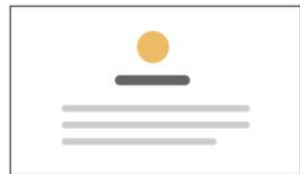
[Next.js](#)는 이러한 단점을 개선하기 위해 **Pre-rendering**을 제공한다

Next.js의 Pre-rendering 특징

Pre-rendering (Using Next.js)

Initial Load:

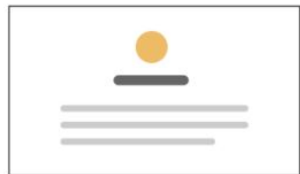
Pre-rendered HTML is displayed



JS loads

Hydration:

React components are initialized and App becomes interactive



If your app has interactive components like `<Link />`, they'll be active after JS loads

Next 앱 - SSR 방식

HTML을 서버에서 미리 렌더링해서 클라이언트로 전송

브라우저는 HTML을 바로 보여줄 수 있음 → **FCP가 빠름**

이후 JS가 실행되면서 React가 동작 (hydration)

Next.js는 JS 실행을 서버쪽에서 실행시켜 렌더링한다. 즉 서버측에서 사전 렌더링된 html을 그대로 브라우저에 보내줌으로써 FCP속도를 빠르게 한다.

사전 렌더링된 html을 보내주면 브라우저가 JS번들을 실행시켜 html과 연결하게 되고 이렇게 되면 상호작용이 가능한 페이지가 되는데 이를 Hydration(수화)이라고 한다.

=> 이렇게 상호작용이 가능하게 된 시점을 TTI(Time to Interactive)라고 한다

Next.js의 Pre-rendering 특징

초기 접속 후 상호작용까지
가능하게 된 이후 페이지 이동
요청이 일어나면
그때는 **CSR**방식과 동일하게
처리된다.
즉 컴포넌트를 교체하여
페이지를 보여주는 형태가 된다
=> **리액트의 장점을 살리면서
단점을 해소**

✅ 언제 서버 렌더링이 더 빠를까?

상황

SSR 유리함?

✅ 정적 콘텐츠 or 첫 페이지 진입 시

👍 유리함

✅ SEO가 중요한 페이지 (블로그, 마케팅 등)

👍 매우 유리

❌ 사용자 상호작용 많은 앱 (채팅, 대시보드 등)

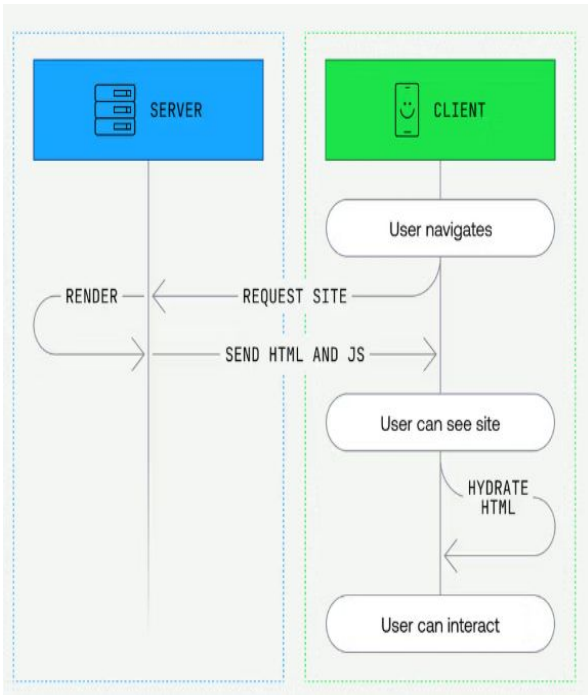
👎 CSR 더 적합

✅ Next.js의 핵심 전략

Next.js는 CSR만 쓰는 React의 한계를 보완해서:

- SSR, SSG, CSR을 상황에 맞게 조합 가능
- 사용자는 빠른 첫 화면을 보고, 이후엔 동적인 React 경험을 가질 수 있음

참고-하이드레이션 (Hydration)이란?



서버 사이드에서 렌더링 된 정적 페이지와 번들링된 JS파일을 클라이언트에게 보낸 뒤, 클라이언트에서 HTML코드와 React인 JS코드를 서로 매칭 시키는 과정을 의미

SSR을 사용하면 미리 렌더링 된 HTML 구성을 pre-rendering한 후, 생성된 HTML 파일을 클라이언트로 전송한다.

이때 클라이언트가 받은 HTML 파일은 EventListener 등과 같은 각종 JavaScript 코드가 적용되지 않은 상태이다. 이 문제를 해결하기 위해 Hydration을 적용하는데,

NextJS 서버가 pre-rendering된 HTML 파일을 클라이언트에게 전송하자마자 번들링 된 JS코드들을 곧바로 클라이언트에게 전송하고, 전송된 JS코드들이 HTML DOM 위에서 한번 더 rendering되어 각각 제 위치에 매칭이 된다.

Next.js SSR 동작 방식



그림 출처:

<https://prod.velog.io/@taetae-5/%EB%82%98%EC%9D%98-%EA%B0%9C%EB%B0%9C%EC%9D%BC%EC%A7%80-2.-Next.js%EC%9D%98-%EB%8D%B0%EC%9D%B4%ED%84%B0%ED%8C%A8%EC%B9%AD-%EB%B0%A9%EC%8B%9D>

Next.js의 Code Splitting(코드분할)

코드 분할은 웹의 첫 페이지가 로딩될 때, 거대한 javascript payload를 보내는 것이 아니라, **번들을 여러 조각으로 조각내어서 처음에 가장 필요한 부분만 전송해 주는 방식**을 의미한다. 코드 분할을 통해 애플리케이션 로드 타임을 줄여준다.

모든 번들(chunk.js)이 하나로 묶이지 않고, 각각 나뉘어 좀 더 효율적으로 자바스크립트 로딩 시간을 개선할 수 있다

코드 분할은 webpack, parcel, rollup 등의 모듈 번들러도 지원하고 있는 기능이지만 **next.js를 사용하면 별도의 설정없이 자동으로 프로젝트에 적용**된다.

(어떻게 동작할까? dynamic import를 사용하여 모듈이 호출될 때만 모듈을 import한다. 즉 사용될 때만 모듈 불러옴)

Next.js의 특징

이외에도 built-in-CSS, Image Optimization, (react의 hot-reload와 같은) fast refresh, 서버리스 함수로 api 엔드포인트를 만드는 API routes 등 의 기능도 지원한다. 이는 react **프로젝트의 성능 향상** 과 더불어 리액트 어플리케이션을 만들 때 필수적으로 고려해야 할 세부사항들을 적당히 추상화 시켜 제공해 줌으로써 DX(developer experience) 또한 높여준다

Next 사용의 단점

- 프레임워크의 공개빌드된 파일들의 폴더가 .next라서 next.js 사용 프로젝트인게 드러난다.
 - 서버 부하가 CSR에 비해 많은 편이다.
 - 페이지 이동할때마다 새로운 html 파일을 불러올 때 화면이 깜박거림. (UX적으로 좋지못함)
-

개발 환경 및 공식 문서

[Next.js](#) 영어문서

<https://nextjs.org/docs>

한글문서

<https://nextjs-ko.org/docs>

node.js 다운로드

[Node.js — Run JavaScript Everywhere \(nodejs.org\)](#)

[1] node.js 설치하기

<http://nodejs.org> 사이트에서 다운로드
=> LTS (안정화 버전), Current(최신버전)

[2] VSCode 설치

<https://code.visualstudio.com/download>

[3] vscode에 확장 플러그인

- JS JSX Snippets
- JavaScript(ES6) code snippets
- ES7 React/Redux/GraphQL/React-Native snippets
- Live Server/Prettier 등..

[5] React Developer Tool 설치

- <https://chrome.google.com/webstore>
-

실습 : my-next-app 프로젝트 생성

터미널을 열어 아래 명령어를 입력한다

npx create-next-app@14 my-next-app

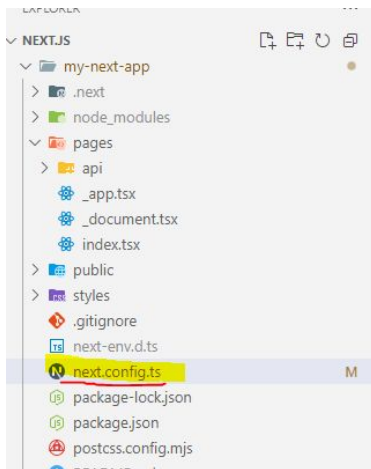
- 페이지 라우터 기능은 14버전
- 앱 라우터 기능은 15버전에서 사용
- 우리는 페이지 라우터 기능부터 익혀보자

프로젝트 생성시 선택할 옵션

```
✓ Would you like to use TypeScript? ... No / Yes
✓ Would you like to use ESLint? ... No / Yes
✓ Would you like to use Tailwind CSS? ... No / Yes
✓ Would you like to use `src/` directory? ... No / Yes
✓ Would you like to use App Router? (recommended) ... No / Yes
? Would you like to customize the default import alias (@/*)? » No / Yes
```

-
- (import alias란?- import할 때 절대 경로로 모듈을 import하도록 도와주는 기능)=>굳이 커스터마이징할 필요는 없음

실습 : my-next-app 프로젝트 생성



```
my-next-app > next.config.ts > nextConfig > reactStrictMode
1 | import type { NextConfig } from 'next';
2 |
3 | const nextConfig: NextConfig = {
4 |   /* config options here */
5 |   reactStrictMode: false,
6 | };
7 |
8 | export default nextConfig;
9 |
```

- 프로젝트가 생성되고 나면 개발 모드로 앱을 실행시키자

npm run dev

- **next.config.mjs** 파일을 열어 **reactStrictMode**를 **false**로 변경 하자

⇒ **true**로 설정하면 개발 모드로 실행했을 때 컴포넌트를 두 번 씩 실행시킨다

```
> my-next-app@0.1.0 dev
> next dev
```

```
▲ Next.js 14.2.30
- Local: http://localhost:3000
```

✓ Starting...

Attention: Next.js now collects completely anonymous telemetry. This information is used to shape Next.js' roadmap and prioritize. You can learn more, including how to opt-out if you'd not like. <https://nextjs.org/telemetry>

브라우저에서 `http://localhost:3000`



1. Get started by editing `pages/index.tsx`.
2. Save and see your changes instantly.



Deploy now

Read our docs



Learn



Examples

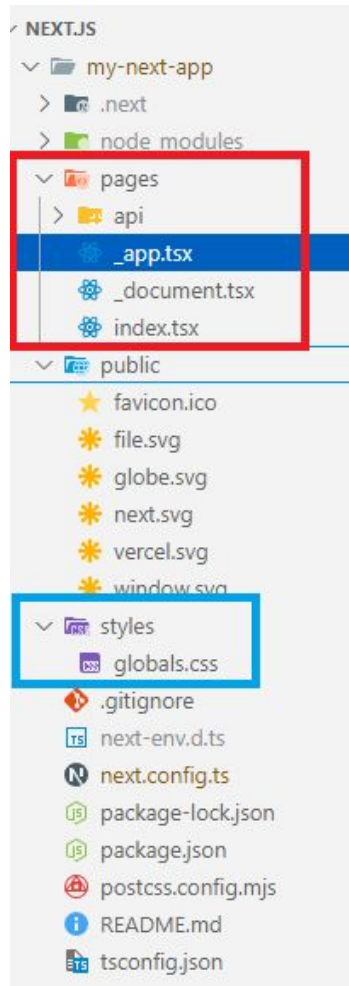


Go to nextjs.org →

기본 프로젝트 구조 (pages 기반 라우팅)

bash

```
my-next-app/
├── node_modules/      # 설치된 npm 패키지
├── public/             # 정적 파일 (이미지, 폰트 등)
│   └── favicon.ico    # 브라우저 탭에 표시될 아이콘
├── pages/              # 라우팅을 위한 페이지 폴더
│   ├── api/           # API 라우팅 폴더
│   │   └── hello.ts   # 예제 API
│   ├── _app.tsx       # 앱 전체 레이아웃 설정
│   ├── _document.tsx  # HTML 문서 구조 제어 (선택적)
│   └── index.tsx      # 루트 페이지 (/)
├── styles/            # CSS, SCSS 등 스타일 관련 파일
│   ├── globals.css    # 전역 스타일 정의
│   └── Home.module.css # CSS 모듈 예제
├── .next/             #  Next.js 빌드 결과물 (자동 생성됨)
├── .gitignore          # Git에서 제외할 파일 목록
├── next.config.js      # Next.js 구성 옵션 (선택적)
├── package.json        # 프로젝트 설정, 스크립트, 의존성
├── tsconfig.json       # TypeScript 설정 파일
└── README.md           # 프로젝트 설명
```



Next.js 프로젝트의 기본 구성 파일

파일명

역할 요약

pages/_app.tsx

모든 페이지의 공통 레이아웃, 상태 관리, CSS 적용 등 앱 레벨 설정

pages/_document.tsx

HTML 문서 구조 커스터마이징 (예: `<html>`, `<body>` 태그 수정)

next.config.mjs

Next.js의 전역 설정 파일 (빌드, 리다이렉트, 이미지 도메인 등)

tsconfig.json

TypeScript 설정 파일

public/ 폴더

정적 파일 (이미지, 파비콘, robots.txt 등)

styles/globals.css

전역 CSS 파일 (주로 `_app.tsx` 에서 import)

React.js와 대응되는 [Next.js](#) 구조

HTML 골격(<html>, <body> 등) →
pages/_document.tsx

React 앱 진입점(App 루트) →
pages/_app.tsx

각 페이지 구성 요소 →
pages/index.tsx,
pages/posts/index.tsx 등

React (create-react-app)	Next.js	설명
public/index.html	pages/_document.tsx	HTML의 <html>, <head>, <body> 구조를 정의 (SSR에서만 사용)
ReactDOM.render(<App />)	pages/_app.tsx	각 페이지 컴포넌트를 감싸는 앱 최상위 컴포넌트
실제 페이지 컴포넌트 (<App />)	pages/index.tsx , pages/about.tsx 등	각각의 라우트에 대응하는 페이지 컴포넌트

◆ React (CRA 기준)

```
public/  
└─ index.html    ← HTML 골격  
src/  
└─ App.tsx      ← 전체 앱 구조  
└─ index.tsx    ← ReactDOM.render(App)
```

◆ Next.js

```
pages/  
└─ _document.tsx ← HTML 골격 정의 (<html>, <head>, <body>)  
└─ _app.tsx      ← 앱 전체를 감싸는 컴포넌트 (레이아웃, 전역 CSS)  
└─ index.tsx     ← 실제 페이지 컴포넌트 (/ 경로에 해당)  
└─ about.tsx     ← 다른 페이지 컴포넌트 (/about 등)
```

Next.js의 Page Routing

Pages Router는 페이지 개념을 기반으로 구축된 파일 시스템 기반 라우터다. 파일이 `pages` 디렉토리에 추가되면 자동으로 경로로 사용할 수 있습니다

`pages` 폴더는 라우팅의 핵심이 되는 폴더로 이 폴더 안에 있는 파일과 폴더의 구조가 그대로 **URL 경로(route)**가 된다. 즉, 파일이 곧 페이지이자 라우트입니다.

`pages` 폴더의 역할

파일/폴더 구조	대응되는 URL 경로	설명
<code>pages/index.tsx</code>	<code>/</code>	루트 페이지 (홈)
<code>pages/about.tsx</code>	<code>/about</code>	정적인 About 페이지
<code>pages/posts/index.tsx</code>	<code>/posts</code>	<code>/posts</code> 경로 페이지
<code>pages/posts/[id].tsx</code>	<code>/posts/1</code> , <code>/posts/abc</code>	동적 라우팅 (id에 따라 다르게)
<code>pages/api/hello.ts</code>	<code>/api/hello</code>	API 엔드포인트 (백엔드 기능)

Page Routing 의 특징

파일 기반 라우팅

- **pages** 폴더 안에 파일을 만들면 자동으로 라우터가 생성된다.
- `index.tsx`는 해당 폴더의 기본 페이지다.

동적 라우팅 지원

- `[param].tsx` 형태로 파일명을 지정하면 URL 파라미터를 받을 수 있다.

API Routes 지원 (**pages/api**)

- `pages/api` 폴더 안은 React 컴포넌트가 아니라 서버에서 실행되는 API 핸들러 함수다.
- 예: `GET /api/posts` → 데이터를 JSON으로 응답 가능

자동 코드 분할

- 페이지 단위로 코드가 분할되어 초기 로딩 속도 최적화에 유리하다.
 - **SSR, SSG, CSR 모두 가능**
=> 각 페이지마다 `getServerSideProps`, `getStaticProps`, `useEffect` 등을 통해 원하는 렌더링 방식을 선택할 수 있다.
-

동적 라우팅

사전에 정확한 세그먼트 이름을 모르고 동적 데이터에서 경로를 생성하려는 경우 요청 시 채워지거나 빌드 시 미리 렌더링되는 동적 세그먼트를 사용할 수 있다.

예

예를 들어, 블로그에는 블로그 게시물에 대한 동적 세그먼트가 있는 `pages/blog/[slug].js` 다음 경로가 포함될 수 있습니다. `[slug]`

```
1 import { useRouter } from 'next/router'
2
3 export default function Page() {
4   const router = useRouter()
5   return <p>Post: {router.query.slug}</p>
6 }
```

Route	Example URL	params
pages/blog/[slug].js	<u>/blog/a</u>	{ slug: 'a' }
pages/blog/[slug].js	/blog/b	{ slug: 'b' }
pages/blog/[slug].js	/blog/c	{ slug: 'c' }

동적 라우팅 - 포괄적 세그먼트

동적 세그먼트는 괄호 안에 줄임표를 추가하여 이후의 모든 세그먼트를 포함하도록 확장할 수 있다 `[...segmentName]`

예를 들어, `pages/shop/[...slug].js`는

`/shop/clothes`, `/shop/clothes/tops`, `/shop/clothes/tops/t-shirts`, 등의 요청도 받는다

Route	Example URL	params
<code>pages/shop/[...slug].js</code>	<code>/shop/a</code>	<code>{ slug: ['a'] }</code>
<code>pages/shop/[...slug].js</code>	<code>/shop/a/b</code>	<code>{ slug: ['a', 'b'] }</code>
<code>pages/shop/[...slug].js</code>	<code>/shop/a/b/c</code>	<code>{ slug: ['a', 'b', 'c'] }</code>

동적 라우팅 - Optional 세그먼트

포괄적인 세그먼트는 매개변수를 이중 대괄호로 포함하여 선택 사항으로 `[...segmentName]` 만들 수 있다

포괄적인 세그먼트와 선택적 포괄적인 세그먼트의 차이점은 선택적 세그먼트의 경우 매개변수가 없는 경로도 일치한다는 것

Route	Example URL	params
<code>pages/shop/[... slug].js</code>	<code>/shop</code>	<code>{ slug: undefined }</code>
<code>pages/shop/[... slug].js</code>	<code>/shop/a</code>	<code>{ slug: ['a'] }</code>
<code>pages/shop/[... slug].js</code>	<code>/shop/a/b</code>	<code>{ slug: ['a', 'b'] }</code>
<code>pages/shop/[... slug].js</code>	<code>/shop/a/b/c</code>	<code>{ slug: ['a', 'b', 'c'] }</code>

Next.js 렌더링 방법

1. SSG (Static Site Generation) : 정적 사이트 생성
2. CSR (Client Side Rendering)
3. SSR (Server Side Rendering)
4. ISR (Incremental Static Regeneration) : 점진적 정적 재생성

모든 페이지에서 사전 렌더링 가능한 부분은 사전 렌더링을
수행한다

<1> SSG - 정적 사이트 생성

웹사이트를 빌드할 때 모든 페이지를 미리 정적으로 생성하는 방식
웹페이지를 **HTML 파일로 미리 생성**해두고, 클라이언트가 요청하면 서버는
이 HTML을 그대로 전달하는 방식

빌드시 API 등으로 부터 데이터를 얻어 페이지를 그려 정적 파일로 생성한다

빌드시 **getStaticProps()** 라는 함수가 호출되며, 그 함수 안에서 API 호출 등을 수행하고,
페이지를 그리는 데 필요한 props를 반환한다

그 뒤, 이 props를 페이지 컴포넌트에 전달해서 화면을 그린다

그린 결과는 정적 파일의 형태로 빌드 결과로 저장한다

페이지에 접근하게 되면 미리 생성한 정적파일을 클라이언트에 보내고 브라우저는
그것을 기반으로 화면을 그린다

브라우저에서 초기화면을 그린 후에는 일반 리액트와 마찬가지로 동작함

<1> SSG

- 더 빠른 페이지 로드를 제공하거나
 - 서버 측 처리를 최소화하고,
 - 검색 엔진 최적화를 개선하고,
 - 내용이 많거나 절대 변경되지 않는 내용(블로그, 포트폴리오, 뉴스 사이트, 문서 등)이 있는 페이지를 업로드하려는 경우 **Next.js**를 사용한 정적 사이트 생성을 사용해야 합니다
-

<1> SSG

- SSG는 유저가 접근 시 가지고 있던 정적파일을 전달하기만 하기때문에 초기 화면을 그리기까지가 빠르다.
 - 하지만, 가지고 있던 데이터가 마지막 빌드 시점의 데이터이기 때문에 stale상태이므로 오래된 데이터가 보여질 가능성이 크기 때문에 **실시간성이 중요한 웹서비스는 맞지 않는 방법**이다.
 - 블로그 등의 서비스가 적절해보인다
 - 또한, **성능이 뛰어나기 때문에 Next.js에서도 SSR보다는 SSG를 권장**한다.
-

<1> SSG - getStaticProps 함수

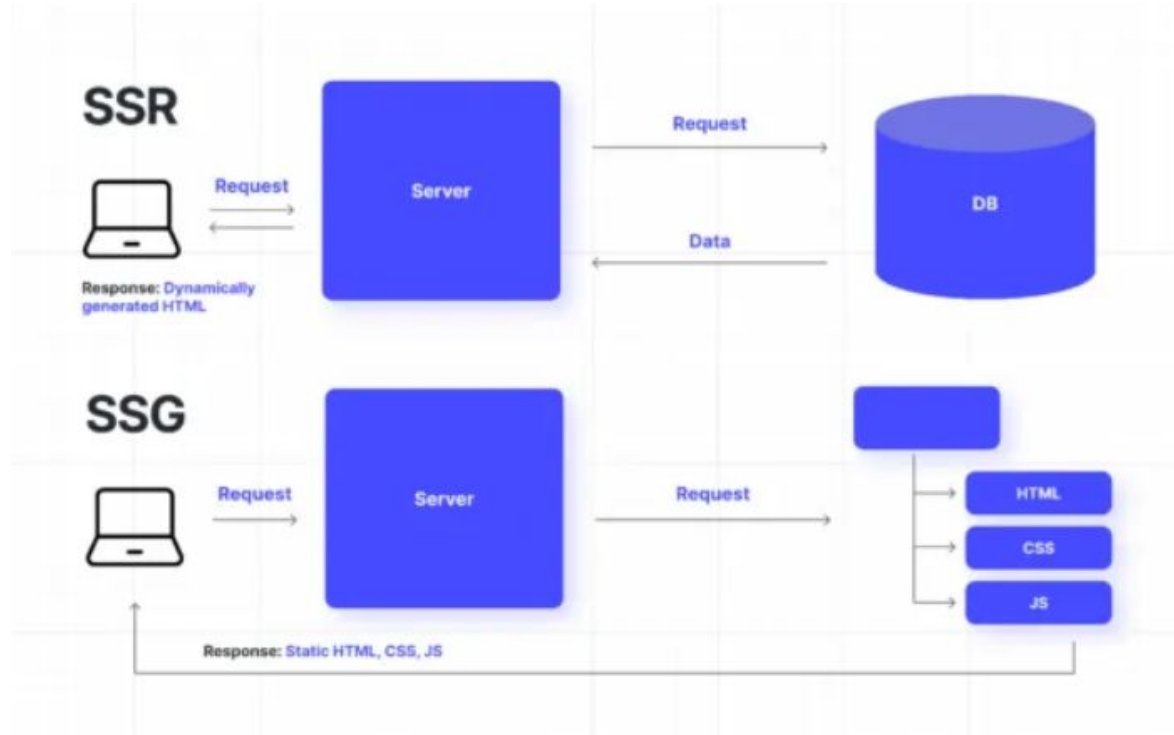
getStaticProps

- 페이지에서 사용할 props를 준비하는 기능을 합니다.
 - **promise**를 반환합니다. (NextJs는 promise가 해결될 때 대기합니다)
 - 클라이언트 측에 절대 실행되지 않기 때문에 어떤 코드든지 실행이 가능합니다. (파일시스템, 데이터베이스 등...)
 - 컴포넌트 함수에서 사용할 **props**를 반환합니다. (반환값은 보통 props 프로퍼티를 설정하며 key이름은 반드시 props여야 합니다. 여기에서 반환되는 props가 컴포넌트 함수의 매개변수로 전달되어 사용됩니다.)
-

<1> SSG - getStaticProps 예제

```
const Component = (props) => {  
  return <List listData={props.listData} />;  
};  
  
export async function getStaticProps(){  
  // fetch 통신처리 처리후 DUMMY_LIST을 가져왔다고 가정  
  const DUMMY_LIST = fetch(...)  
  
  return {  
    props:{  
      listData : DUMMY_LIST,  
    }  
  };  
}
```

SSR vs SSG



<2> SSR- 서버측 렌더링

SSR(Server Side Rendering) 은 서버 사이드 렌더링의 줄임말로, 브라우저가 요청한 시점에 서버가 HTML을 생성해 클라이언트에 전달하는 방식

- SSR에서 사용자가 웹 페이지를 요청하면
 - Next.js 서버는 요청을 처리하고,
 - 데이터를 가져오고,
 - 서버에서 페이지를 조립하고, 완전히 렌더링된 웹 페이지가 사용자에게 전송된다.
 - 렌더링된 페이지는 실시간 콘텐츠를 표시하도록 개인화하거나 업데이트할 수 있다.
 - SSR을 사용하는 웹 페이지의 예로는 사용자별 대시보드가 있다.
-

<2> SSR의 장점

- **실시간 업데이트 및 안전한 데이터:** SSR을 사용하면 Next.js 애플리케이션에서 실시간 업데이트를 통해 안전한 사용자 인증을 수행할 수 있으므로 데이터 개인 정보 보호와 웹사이트 유연성이 보장된다.
 - **향상된 SEO:** SSR은 완전히 렌더링된 HTML 페이지를 제공하여 더 나은 검색 엔진 최적화를 보장한다.
 - **서버 측 데이터 가져오기:** 애플리케이션에 서버에서 데이터 처리가 필요한 경우 SSR을 사용하면 서버 측 데이터나 외부 API를 원활하게 가져올 수 있다.
-

<2> SSR의 단점

- **시간이 많이 소요되는 구현:** SSR을 사용하면 서버 측 코드 관리, 데이터 가져오기 등의 기능이 필요하므로 시간이 많이 소요될 수 있다.
 - **서버 부하 증가:** SSR은 특히 사용자 요청량이 많을 때 서버 리소스를 많이 소모하여 Next.js 애플리케이션의 확장성과 성능에 영향을 미칠 수 있다.
-

<2> SSR - getServerSideProps 함수

getServerSideProps

- nextJs는 사전 렌더링 중 getServerSideProps 함수를 발견하면 컴포넌트 함수 호출전에 getServerSideProps 함수를 먼저 호출 합니다.
 - 페이지에서 사용할 props를 준비하는 기능을 합니다.
 - **promise**를 반환합니다. (NextJs는 promise가 해결될 때 까지 기다린다)
 - 클라이언트 측에 절대 실행되지 않기 때문에 어떤 코드든지 실행이 가능합니다. (파일시스템, 데이터베이스 등...)
 - revalidate는 없고 매개변수가 존재하는데 매개변수에는 요청객체, 응답객체가 있고 미들웨어에서 함께 작업 합니다. 요청객체를 통해 헤더와 바디에도 접근 가능합니다.
 - 헤더와 바디 접근이 가능해서 인증작업, 세션 쿠키를 할 때 도움이 될 수 있습니다.
-

<2> SSR - getServerSideProps 예제

```
// pages/about.js
import React from "react";
const About = ({ serverData }) => (
  <div>
    <h1>정보 페이지</h1>
    <p>{serverData}</p>
  </div>
);
export default About;
export async function getServerSideProps () {
  const serverData = "이 데이터는 서버 측에서 가져옵니다." ;
  return {
    props : {
      serverData,
    },
  };
}
```

<3> ISR - 증분 정적 재생성

ISR(Incremental Static Regeneration, 증분 정적 재생성) 은 SSG(Static Site Generation) 방식에서 일정한 주기마다 페이지를 자동으로 재생성 하여 최신 데이터를 반영하는 기능을 말한다

ISR의 동작 원리

- Next.js에서 SSG 방식으로 사전 렌더링된 페이지는 빌드 타임에 생성된 이후 내용이 고정되므로, 이후에 발생하는 데이터 변화가 반영되지 않는다.
 - 이 경우 ISR을 사용하면 정적 페이지에 유효기한을 설정하여 특정 주기마다 페이지가 다시 생성되도록 설정할 수 있다.
 - 예를 들어, 60초마다 재생성하도록 설정하면, 60초 동안 동일한 페이지를 제공하다가 이 시간이 지나면 다음 접속 시 새로운 데이터를 반영한 페이지로 갱신하여 사용자에게 전달한다.
-

<3> ISR - 증분 정적 재생성

- 페이지를 빌드 타임에 정적으로 생성하지만,
- 일정 시간이 지나면 백그라운드에서 새로 갱신함.
- 사용자는 항상 빠른 정적 페이지를 보고, 새 콘텐츠는 지연 없이 점진적으로 반영됨.

ISR의 예시

예를 들어, 인덱스 페이지에 추천 도서 목록이 포함되어 있고 이 목록이 매번 동일한 책들을 보여주기보다는 주기적으로 업데이트되어 다양한 책을 추천하는 것이 바람직하다고 가정해보자. 이전에 인덱스 페이지는 `getStaticProps` 함수를 통해 SSG 방식으로 설정된 상태여서, 사용자가 새로고침을 하더라도 동일한 책 목록이 반복해서 노출되었다. ISR 을 적용하면, 이 추천 도서 목록을 일정 주기마다 새로운 내용으로 업데이트할 수 있다.

<3> ISR 동작 과정

1. 초기 빌드 타임 생성: 빌드 타임에 SSG 방식으로 페이지가 한 번 생성되며, 사용자가 이 페이지를 요청하면 생성된 정적 페이지가 즉시 반환된다.
 2. 갱신 주기 설정: `revalidate` 로 설정한 시간이 지나기 전까지는 캐시된 기존 페이지가 반복적으로 제공된다.
 3. 갱신 주기 만료 시 첫 요청 처리: 설정된 시간이 지나고 첫 번째 요청이 들어오면 ISR 방식이 동작하여, Next.js는 우선 캐시된 페이지를 반환하고 서버는 백그라운드에서 새로운 페이지를 생성한다.
 4. 새 페이지로 갱신: 이후의 요청부터는 최신 데이터가 반영된 새 버전의 페이지가 빠르게 제공된다.
-

<3> ISR - getStaticProps+revalidate

1. getStaticProps

- 정적 페이지 생성 시 데이터를 가져오는 함수
- SSG의 핵심 함수이며, 여기에 revalidate 옵션을 추가하면 ISR이 적용됩니다.

tsx

```
export async function getStaticProps(context) {  
  // 데이터를 API나 DB 등에서 가져옴  
  const data = await fetchData();  
  
  return {  
    props: {  
      data, // 페이지에 전달할 props  
    },  
    revalidate: 60, // ISR: 60초마다 백그라운드에서 갱신  
  };  
}
```

<3> ISR - getStaticProps+revalidate

주요 리턴 값

리턴 속성

설명

`props`

페이지 컴포넌트에 전달될 데이터

`revalidate`

초 단위 시간. 이 시간이 지나면 백그라운드에서 페이지 재생성

`notFound`

`true` 일 경우 404 페이지 렌더링

`redirect`


리다이렉트 설정 가능

<3> ISR -getServerPaths

동적 경로 (Dynamic Route)를 사용할 때, 어떤 경로를 정적으로 생성할지 지정하는 함수

```
export async function getStaticPaths() {  
  const posts = await fetchPostIds();  
  
  const paths = posts.map(id => ({  
    params: { id: id.toString() },  
  }));  
  
  return {  
    paths,  
    fallback: 'blocking', // 존재하지 않는 경로는 서버에서 생성 (ISR과 공한 좋음)  
  };  
}
```

<3> ISR -getServerPaths

 fallback 의 옵션

옵션	설명
false	paths 에 명시된 것만 정적 생성, 나머지는 404
true	나머지는 **브라우저에서 "로딩 중"**으로 처리 후 렌더링
blocking	나머지는 서버에서 완전히 생성 후 사용자에게 보여줌 (추천 방식)

언제 ISR 을 쓰면 좋을까?

상황	추천 여부
블로그, 뉴스, 상품 페이지처럼 자주 바뀌지 않지만 업데이트가 필요	✅ 매우 적합
사용자별 맞춤 페이지 (예: 로그인된 사용자 정보)	❌ 부적합 (SSR 또는 CSR 사용 권장)

<3> ISR - 예제

```
// pages/posts/[id].tsx

import { GetStaticPaths, GetStaticProps } from 'next';
import { fetchPost, fetchAllPostIds } from '../lib/posts';

export default function Post({ post }: { post: any }) {
  return (
    <div>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </div>
  );
}

// getStaticPaths는 어떤 동적 경로를 미리 생성할지 결정
export const getStaticPaths = async () => {
  const ids = await fetchAllPostIds();
  const paths = ids.map(id => ({ params: { id } }));

  return {
    paths,
    fallback: 'blocking', // 없는 페이지는 서버에서 생성 후 캐시
  };
};
```

```
// getStaticProps는 각 경로에 필요한 데이터를 가져옴
export const getStaticProps: GetStaticProps = async (context) => {
  const id = context.params?.id as string;
  const post = await fetchPost(id);

  return {
    props: { post },
    revalidate: 60, // ISR: 60초마다 백그라운드에서 페이지 재생성
  };
};
```

/posts/1 요청 시, 정적으로 미리 생성된 HTML이 즉시 응답됨.

revalidate: 60 → 이 페이지는 최소 60초 간 유효함.

60초 이후 누군가 요청하면: 기존 HTML로 응답하고,
백그라운드에서 새 HTML 생성 및 캐시 갱신.

다음 요청부터는 새 HTML을 사용.

Next.js의 페이지와 데이터 취득

Next.js에서는 페이지에서 구현하는 함수나 해당 함수의 반환값에 따라, pages의 렌더링 방법이 달라진다.

종류	데이터 취득에 사용하는 주요 함수	데이터 취득 시점	보충 설명
SSG	<code>getStaticProps</code>	<u>빌드 시(SSG)</u>	데이터 취득을 전혀 수행하지 않는 경우도 SSG에 해당
SSR	<code>getServerSideProps</code>	<u>사용자 요청 시(서버 사이드)</u>	<code>getInitialProps</code> 를 사용해도 SSR
ISR	revalidate를 반환하는 <code>getStaticProps</code>	<u>빌드 시(ISR)</u>	ISR은 배포 후에도 백그라운드 빌드가 실행된다.
CSR	그 밖의 임의의 함수(useSWR 등)	<u>사용자 요청 시(브라우저)</u>	CSR은 SSG/SSR/ISR과 동시에 사용 가능

pages는 그 종류에 따라 데이터 취득에 사용할 수 있는 함수가 다르고, 페이지 컴포넌트에서 모든 표시 부분을 구현할 필요는 없다.

페이지 사이에서 공통으로 사용하는 코드나 UI부분은 pages 디렉터리 밖에서 정의하고 임포트해서 사용이 가능하다.

실습 - 쇼핑몰

쇼핑몰 구축시 [Next.js](#)의 어떤 방식을 선택하면 좋을까?

페이지	설명	렌더링 방식	이유
 홈 (Home)	신상품 목록, 베스트 상품	SSG	자주 바뀌지 않고 모든 사용자에게 동일한 정보
 상품 상세 (Product Detail)	제품 ID에 따라 상세 보기	ISR	제품 정보는 거의 정적이지만 재고/가격은 가끔 바뀜
 검색 (Search)	사용자가 키워드로 검색	SSR	검색 결과는 사용자 입력에 따라 실시간 반응 필요
 장바구니 / 결제	로그인 후 사용자별 상태	CSR	개인 상태 기반. 브라우저 내에서 상태 관리
 마이페이지	주문 내역, 정보 수정 등	SSR or CSR	로그인 필요, 사용자마다 달라짐

참고 사이트

<https://nextjs.org/learn/pages-router>

<https://next-learn-starter.vercel.app/>

<https://velog.io/@codns1223/Nextjs-Next.js%EB%9E%80>

<https://subtlething.tistory.com/115>

<https://velog.io/@taetae-5/posts>

<https://medium.com/@chrisebuberoland/static-site-generation-ssg-vs-server-side-rendering-in-next-js-debf43f4bb7f>
