

NextJS란?

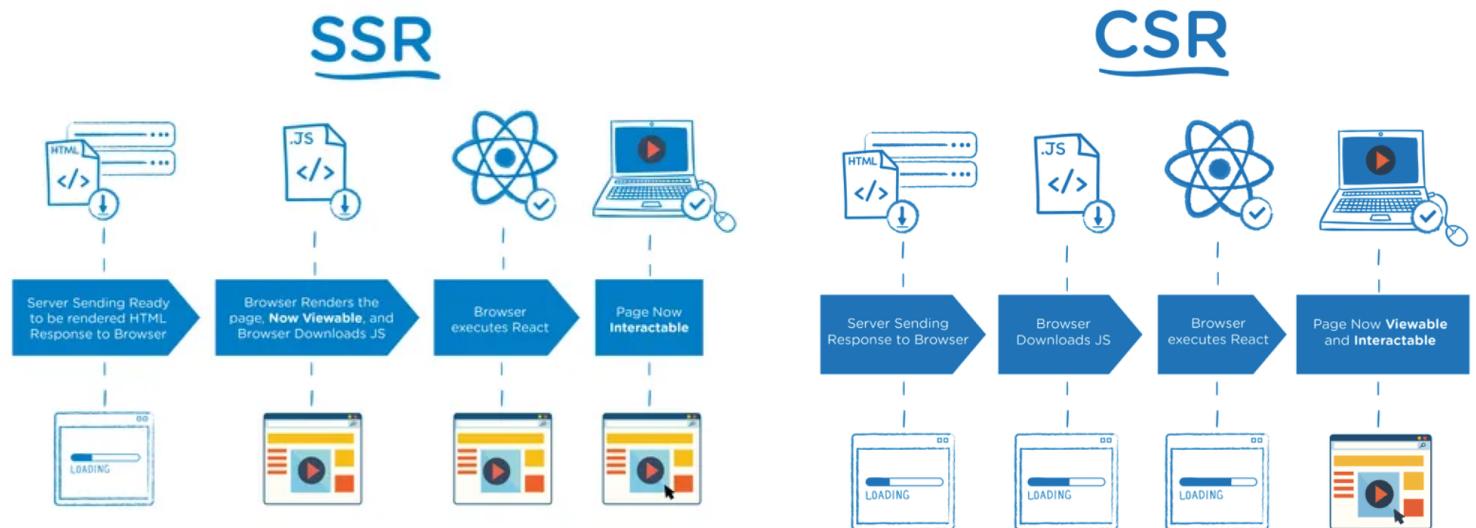
Next JS란?

React의 SSR(server side rendering)을 쉽게 구현할 수 있게 도와 주는 간단한 프레임워크입니다.
(리액트는 라이브러리)

리액트로 개발할 때 SPA(single Page Application)을 이용하면 CSR(Client Side Rendering)을 하기 때문에 좋은 점도 있지만 단점도 있는데 그 부분이 바로 검색엔진 최적화(SEO) 부분입니다.
Client Side Rendering을 하면 첫페이지에서 빈 html을 가져와서 JS파일을 해석하여 화면을 구성하기 때문에 포털 검색에 거의 노출 될 일이 없습니다.

하지만 Next.js에서는 Pre-Rendering을 통해서 페이지를 미리 렌더링 하며 완성된 HTML을 가져오기 때문에 사용자와 검색 엔진 크롤러에게 바로 렌더링 된 페이지를 전달할 수 있게 됩니다.

리액트에서도 SSR을 지원하기 위해 구현하기에 굉장히 복잡하기 때문에 Next.js를 통해서 이 문제를 해결해주게 됩니다.



Server Side Rendering

- 클라이언트 대신 서버에서 페이지를 준비하는 원리입니다.
 - 원래 리액트에서는 클라이언트 사이드 렌더링하기 때문에 서버에 영향을 미치지 않고, 서버에서 클라이언트로 응답해서 보낸 html도 거의 비어있습니다.
- => 이 방식은 서버에서 데이터를 가져올 때 자연 시간 발생으로 UX 측면에서 좋지 않을 수 있습니다.

=> 검색 엔진에 검색 시 웹크롤링이 동작할 때 내용을 제대로 가져와 읽을 수 없기에 검색엔진 최적화에 문제가 된다.

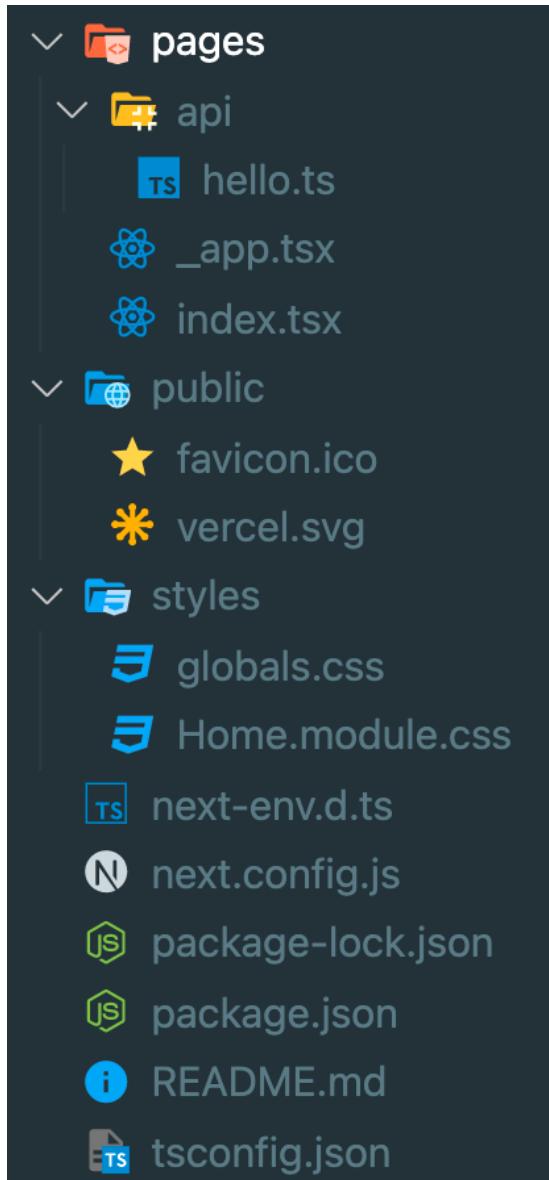
- Next.js에서는 서버 사이드 렌더링을 이용하므로 사용자와 검색 엔진 크롤러에게 바로 렌더링 된 페이지를 전달 할 수 있어서 검색엔진 최적화에 좋은 영향을 줍니다.

설치 방법

```
npx create-next-app@latest  
# or  
yarn create next-app
```

```
npx create-next-app@latest --typescript  
# or  
yarn create next-app --typescript
```

NextJS 기본 파일 구조



pages

- 이 폴더 안에 페이지들을 생성 합니다.
- index.tsx가 처음 "/" 페이지로 됩니다.
- _app.tsx 는 공통되는 레이아웃을 작성합니다. 모든 페이지에 공통으로 들어가는 걸 넣어주려면 여기에 넣어주시면 됩니다. (url을 통해 특정 페이지에 진입하기 전 통과하는 인터셉터 페이지입니다.)
- 만약 about이라는 페이지를 만드시려면 pages 폴더 안에 about.tsx를 생성해주시면 됩니다.

public

- 이미지 같은 정적(static) 에셋들을 보관합니다.

styles

- 말 그래도 스타일링을 처리해주는 폴더입니다.
- 모듈(module) css는 컴포넌트 종속적으로 스타일링하기 위한 것이며, 확장자 앞에 module을 붙여줘야 합니다.

next.config.js

- Nextjs는 웹팩을 기본 번들러로 사용합니다.
- 그래서 웹팩에 관한 설정들을 이 파일에서 해줄수 있습니다.

Pre-rendering

NextJS는

모든 페이지를 pre-render 합니다. 이 pre-render한다는 의미는 모든 페이지를 위한 HTML을 Client사이드에서 자바스크립트로 처리하기 전, "사전에" 생성한다는 것입니다.
이렇게 하기 때문에 SEO 검색엔진 최적화가 좋아집니다.

Pre Render 테스트 !!!

자바스크립트 Disable

<https://developer.chrome.com/docs/devtools/javascript/disable/>

보통 React 사이트 들어가기

<https://create-react-app.examples.vercel.com/>

NextJS 사이트 들어가기

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

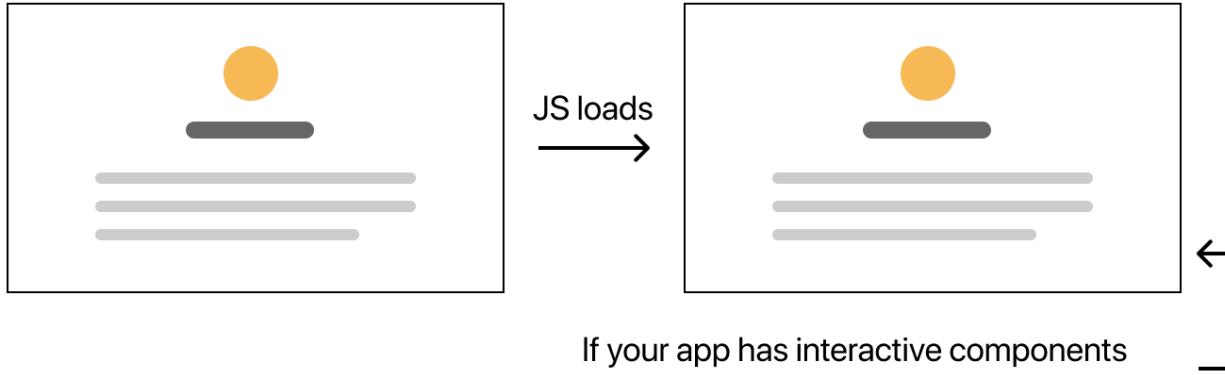
Summary !!!

Pre-rendering (Using Next.js)

Initial Load:

Pre-rendered HTML is displayed

Hydration: React components are initialized and App becomes interactive

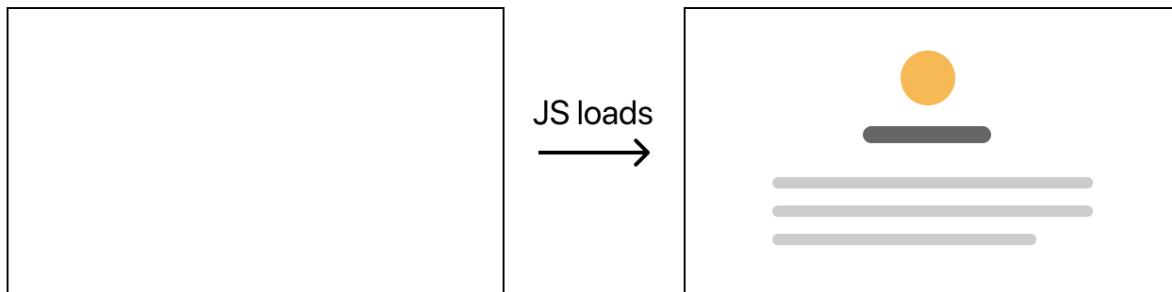


No Pre-rendering (Plain React.js app)

Initial Load:

App is not rendered

Hydration: React components are initialized and App becomes interactive



Data Fetching

Nextjs에서 데이터를 가져오는 방법은...

여러가지가 있습니다. 그래서 애플리케이션의 사용 용도에 따라서 다른 방법을 사용해주면 됩니다.

보통 리액트에서는 데이터를 가져올 때 useEffect안에서 가져옵니다. 하지만 Nextjs에서는 다른 방법을 사용해서 가져오는데 하나씩 봄보겠습니다.

getStaticProps

Static Generation으로 빌드(build)할 때 데이터를 불러옵니다.(미리 만들어줌)

getStaticPaths

Static Generation으로 데이터에 기반하여 pre-render시 특정한 동적 라우팅 구현(pages/post/[id].js)

getServerSideProps

Server Side Rendering으로 요청이 있을 때 데이터를 불러옵니다.

```
export async function getStaticProps(context) {
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

getStaticProps

- getStaticProps 함수를 async로 export 하면, getStaticProps에서 리턴되는 props를 가지고 페이지를 pre-render 합니다. build time에 페이지를 렌더링 합니다.

```
// posts will be populated at build time by getStaticProps()
function Blog({ posts }) {
```

```

function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>{post.title}</li>
      )))
    </ul>
  )
}

// This function gets called at build time on server-side.
// It won't be called on client-side, so you can even do
// direct database queries.
export async function getStaticProps() {
  // Call an external API endpoint to get posts.
  // You can use any data fetching library
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // By returning { props: { posts } }, the Blog component
  // will receive `posts` as a prop at build time
  return {
    props: {
      posts,
    },
  }
}

export default Blog

```

getStaticProps를 사용해야 할 때

- 페이지를 렌더링하는 데 필요한 데이터는 사용자의 요청보다 먼저 build 시간에 필요한 데이터를 가져올 때
- 데이터는 Headless CMS에서 데이터를 가져올 때.
- 데이터를 공개적으로 캐시할 수 있을 때(사용자별 아님).
- 페이지는 미리 렌더링되어야 하고(SEO의 경우) 매우 빨라할 때.(getStaticProps는 성능을 위해 CDN에서 캐시할 수 있는 HTML 및 JSON 파일을 생성합니다.)

```
export async function getStaticPaths() {
```

```
return {
  paths: [
    { params: { ... } }
  ],
  fallback: true // false or 'blocking'
};
}
```

getStaticPaths

- 동적 라우팅이 필요할 때 getStaticPaths로 경로 리스트를 정의하고, HTML에 build 시간에 렌더 됩니다.
- Nextjs는 pre-render에서 정적으로 getStaticPaths에서 호출하는 경로들을 가져옵니다.

paths

- 어떠한 경로가 pre-render 될지를 결정합니다.
- 만약 pages/posts/[id].js 이라는 이름의 동적 라우팅을 사용하는 페이지가 있다면 아래와 같이 됩니다.

```
return {
  paths: [
    { params: { id: '1' } },
    { params: { id: '2' } }
  ],
  fallback: ...
}
```

- 빌드하는 동안 /posts/1과 /posts/2를 생성하게 됩니다.

params

- 페이지 이름이 pages/posts/[postId]/[commentId] 라면, params은 postId와 commentId입니다.
- 만약 페이지 이름이 pages/...slug] 와 같이 모든 경로를 사용한다면, params는 slug 가 담긴 배열이어야한다. ['postId', 'commentId']

fallback

- false 라면 getStaticPaths로 리턴되지 않는 것은 모두 404 페이지가 됩니다.

- true 라면 getStaticPaths로 리턴되지 않는 것은 404로 뜨지 않고 , fallback 페이지가 뜨게 됩니다.

```
// If the page is not yet generated, this will be displayed
// initially until getStaticProps() finishes running
if (router.isFallback) {
  return <div>Loading...</div>
}
```

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

function Post({ post }) {
  // Render post...
}

// This function gets called at build time
export async function getStaticPaths() {
  // Call an external API endpoint to get posts
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // We'll pre-render only these paths at build time.
  // { fallback: false } means other routes should 404.
  return { paths, fallback: false }
}

// This also gets called at build time
export async function getStaticProps({ params }) {
  // params contains the post `id`.
  // If the route is like /posts/1, then params.id is 1
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  // Pass post data to the page via props
```

```
    return { props: { post } }

}

export default Post
```

```
export async function getServerSideProps(context) {
  return {
    props: {}, // will be passed to the page component as props
  }
}
```

getServerSideProps

- getServerSideProps 함수를 async로 export 하면, Next는 각 요청마다 리턴되는 데이터를 getServerSideProps로 pre-render합니다.

```
function Page({ data }) {
  // Render data...
}

// This gets called on every request
export async function getServerSideProps() {
  // Fetch data from external API
  const res = await fetch(`https://.../data`)
  const data = await res.json()
```

```
// Pass data to the page via props
return { props: { data } }
}

export default Page
```

getServerSideProps를 사용해야 할 때

- 요청할 때 데이터를 가져와야하는 페이지를 미리 렌더해야 할 때 사용합니다. 서버가 모든 요청에 대한 결과를 계산하고, 추가 구성없이 CDN에 의해 결과를 캐시할 수 없기 때문에 첫번째 바이트까지의 시간은 getStaticProps보다 느립니다.

TypeScript 란?

참조: https://www.tutorialspoint.com/typescript/typescript_overview.htm

TypeScript 가 나오게 된 배경..

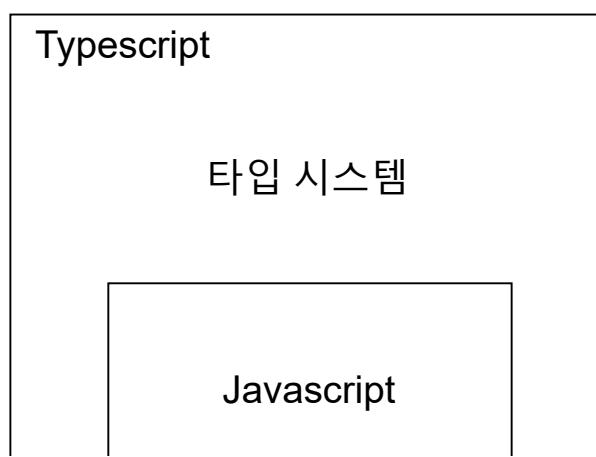
JavaScript는 원래 클라이언트 측 언어로 도입되었습니다. 그런데 Node.js의 개발로 인해서 JavaScript를 클라이언트 측 뿐만이 아닌 서버 측 기술로도 활용되게 만들었습니다. 그러나 JavaScript 코드가 커질수록 소스 코드가 더 복잡해져서 코드를 유지 관리하고 재사용하기가 어려워졌습니다. 더욱이 Type 검사 및 컴파일 시 오류 검사의 기능을 수용하지 못하기 때문에 JavaScript가 본격적인 서버 측 기술로 엔터프라이즈 수준에서 성공하지 못합니다. 이 간극을 메우기 위해 TypeScript가 제시되었습니다.

TypeScript 란?

타입스크립트는 자바스크립트에 타입을 부여한 언어입니다. 자바스크립트의 확장된 언어라고 볼 수 있습니다. 타입스크립트는 자바스크립트와 달리 브라우저에서 실행 하려면 파일을 한번 변환해주어야 합니다. 이 변환 과정을 우리는 컴파일(complile)이라고 부릅니다.



<https://www.typescripttutorial.net/typescript-tutorial/what-is-typescript/>



Type System

- 개발 환경에서 에러를 잡는 걸 도와줍니다.
- type annotations를 사용해서 코드를 분석할 수 있습니다.
- 오직 개발 환경에서만 활성화 됩니다.
- 타입 스크립트와 성능 향상과는 관계가 없습니다.

TypeScript 사용하는 이유 ?

- TypeScript는 JavaScript 코드를 단순화하여 더 쉽게 읽고 디버그할 수 있도록 합니다.
- TypeScript는 오픈 소스입니다.
- TypeScript는 정적 검사와 같은 JavaScript IDE 및 사례를 위한 매우 생산적인 개발 도구를 제공합니다.
- TypeScript를 사용하면 코드를 더 쉽게 읽고 이해할 수 있습니다.
- TypeScript를 사용하면 일반 JavaScript보다 크게 개선할 수 있습니다.
- TypeScript는 ES6(ECMAScript 6)의 모든 이점과 더 많은 생산성을 제공합니다.
- TypeScript는 코드 유형 검사를 통해 JavaScript를 작성할 때 개발자가 일반적으로 겪는 고통스러운 버그를 피하는 데 도움이 될 수 있습니다.

Nextjs와 Typescript 만들 앱 소개



[Your Name]

[Your Self Introduction]

(This is a sample website - you'll be building a site like this in our [Next.js tutorial](#).)

Blog

[When to Use Static Generation v.s. Server-side Rendering](#)

January 2, 2020

간단한 블로그 앱

nextjs 공식 사이트 Documentation에서 nextjs를 배우기 위해 만드는 앱.

블로그 포스트 내용은 md 파일로 작성.

메인 페이지 UI 만들기(마크다운 파일 생성)

```
const Home: NextPage = () => {
  return (
    <div>
      <Head>
        <title>Your Name</title>
      </Head>
      <section>
        <p>[Your Self Introduction]</p>
        <p>
          (This is a website)
        </p>
      </section>
      <section >
        <h2>Blog</h2>
        <ul>

        </ul>
      </section>
    </div>
  )
}

export default Home
```

```
const Home: NextPage = () => {
  return (
    <div>
      <Head>
        <title>Your Name</title>
      </Head>
      <section className={homeStyles.headingMd}>
        <p>[Your Self Introduction]</p>
        <p>
          (This is a website)
        </p>
      </section>
      <section className={`${homeStyles.headingMd} ${homeStyles.padding1px}`}>
        <h2 className={homeStyles.headingLg}>Blog</h2>
        <ul className={homeStyles.list}>

        </ul>
      </section>
    </div>
  )
}
```

md 파일 안에 포스트 생성하기

markdown 파일이란..?

Markdown은 텍스트 기반의 마크업언어로 쉽게 쓰고 읽을 수 있으며 HTML로 변환이 가능합니다. 특수기호와 문자를 이용한 매우 간단한 구조의 문법을 사용하여 웹에서도 보다 빠르게 컨텐츠를 작성하고 보다 직관적으로 인식할 수 있습니다. 마크다운이 최근 각광받기 시작한 이유는 깃헙(<https://github.com>)에서 사용하는 README.md 덕분입니다. 마크다운을 통해서 설치 방법, 소스코드 설명, 이슈 등을 간단하게 기록하고 가독성을 높일 수 있다는 강점이 부각되면서 점점 여러 곳으로 퍼져가게 되고 있습니다.

```
---
title: "When to Use Static Generation v.s. Server-side Rendering"
date: "2020-01-02"
---

We recommend using Static Generation (with and without data)
whenever possible because your page can be built once and served by
CDN, which makes it much faster than having a server render the page on
every request.

You can use Static Generation for many types of pages, including:

- Marketing pages
- Blog posts
- E-commerce product listings
- Help and documentation

You should ask yourself: "Can I pre-render this page ahead of a
user's request?" If the answer is yes, then you should choose Static
Generation.

On the other hand, Static Generation is not a good idea if you
cannot pre-render a page ahead of a user's request. Maybe your page
shows frequently updated data, and the page content changes on every
request.

In that case, you can use Server-Side Rendering. It will be slower,
but the pre-rendered page will always be up-to-date. Or you can skip
pre-rendering and use client-side JavaScript to populate data.
```

```
---
title: "Two Forms of Pre-rendering"
date: "2020-01-01"
---

Next.js has two forms of pre-rendering: Static Generation and Server-
side Rendering. The difference is in when it generates the HTML for a
page.
```

- **Static Generation** is the pre-rendering method that generates the HTML at **build time**. The pre-rendered HTML is then reused on each request.
- **Server-side Rendering** is the pre-rendering method that generates the HTML on **each request**.

Importantly, Next.js lets you **choose** which pre-rendering form to use for each page. You can create a "hybrid" Next.js app by using Static Generation for most pages and using Server-side Rendering for others.

posts 폴더 생성

파일 생성 후
마크다운 작성하기

마크다운 파일을 데이터로 추출하기

markdown file

데이터

```
---
title: "When to Use Static Generation v.s. Server-side Rendering"
date: "2020-01-02"
---

We recommend using Static Generation (with and without data) whenever possible because
your page can be built once and served by CDN, which makes it much faster than having a
server render the page on every request.

You can use Static Generation for many types of pages, including:

- Marketing pages
- Blog posts
- E-commerce product listings
- Help and documentation

You should ask yourself: "Can I pre-render this page ahead of a user's request?" If the
answer is yes, then you should choose Static Generation.

On the other hand, Static Generation is not a good idea if you cannot pre-render a page
ahead of a user's request. Maybe your page shows frequently updated data, and the page
content changes on every request.

In that case, you can use Server-Side Rendering. It will be slower, but the pre-rendered
page will always be up-to-date. Or you can skip pre-rendering and use client-side JavaScript
to populate data.
```

Blog

[When to Use Static Generation v.s. Server-side Rendering](#)

→ January 2, 2020

[Two Forms of Pre-rendering](#)

January 1, 2020

post에 사용 할 함수들을
만들 폴더와 파일 생성

lib/posts.ts

```
import fs from 'fs'
import path from 'path'
import matter from 'gray-matter'

const postsDirectory = path.join(process.cwd(), 'posts')
console.log('process.cwd()', process.cwd());
// /Users/johnahn/Downloads/next-typescript
console.log('postsDirectory', postsDirectory);
// /Users/johnahn/Downloads/next-typescript/posts

export function getSortedPostsData() {
  // Get file names under /posts
  const fileNames = fs.readdirSync(postsDirectory)
```

```
fileContents ---
title: "Two Forms of Pre-rendering"
date: "2020-01-01"
---

Next.js has two forms of pre-rendering: inStatic Generation and inSSR.
```

```

console.log('fileNames',fileNames);
// fileNames [ 'pre-rendering.md', 'ssg-ssr.md' ]
const allPostsData = fileNames.map(fileName => {
  // Remove ".md" from file name to get id
  const id = fileName.replace(/\.\md$/, '')

  // Read markdown file as string
  const fullPath = path.join(postsDirectory, fileName)
  const fileContents = fs.readFileSync(fullPath, 'utf8')

  // Use gray-matter to parse the post metadata section
  const matterResult = matter(fileContents)

  // Combine the data with the id
  return {
    id,
    ...(matterResult.data as { date: string; title: string })
  }
})
// Sort posts by date
return allPostsData.sort((a, b) => {
  if (a.date < b.date) {
    return 1
  } else {
    return -1
  }
})
}

```

Next.js has two forms of pre-rendering: **Static Generation** and **when** it generates the HTML for a page.

- **Static Generation** is the pre-rendering method that generates the HTML is then reused on each request.
- **Server-side Rendering** is the pre-rendering method that generates the HTML is then reused on each request.

Importantly, Next.js lets you **choose** which pre-rendering form to use in your Next.js app by using Static Generation for most pages and using Server-side Rendering for specific pages.

```

'matterResult' {
  content: '\n' +
    'Next.js has two forms of pre-rendering: **Static Generation** and **when** it generates the HTML for a page.\n' +
    '\n' +
    '- **Static Generation** is the pre-rendering method that generates the HTML is then _reused_ on each request.\n' +
    '- **Server-side Rendering** is the pre-rendering method that generates the HTML is then _reused_ on each request.\n' +
    '\n' +
    'Importantly, Next.js lets you **choose** which pre-rendering form to use in your Next.js app by using Static Generation for most pages and using Server-side Rendering for specific pages.' +
  'data: { title: "Two Forms of Pre-rendering", date: "2020-01-01" },' +
  'isEmpty: false,' +
}

```

gray-matter

<https://www.npmjs.com/package/gray-matter>

npm install --save gray-matter