Data Fetching

This document is for Next.js versions 9.3 and up. If you're using older versions of Next.js, refer to our previous documentation.

▼ **Examples**

- WordPress Example (Demo)
- Blog Starter using markdown files (Demo)
- DatoCMS Example (Demo)
- TakeShape Example (Demo)
- Sanity Example (Demo)
- Prismic Example (Demo)
- Contentful Example (Demo)
- Strapi Example (Demo)
- Agility CMS Example (Demo)
- Cosmic Example (Demo)
- ButterCMS Example (Demo)
- Storyblok Example (Demo)
- GraphCMS Example (Demo)
- Kontent Example (Demo)
- Static Tweet Demo

In the Pages documentation, we've explained that Next.js has two forms of pre-rendering: **Static Generation** and **Server-side Rendering**. In this page, we'll talk in depth about data fetching strategies for each case. We recommend you to read through the Pages documentation first if you haven't done so.

We'll talk about the three unique Next.js functions you can use to fetch data for pre-rendering:

- `getStaticProps` (Static Generation): Fetch data at **build time**.
- `getStaticPaths` (Static Generation): Specify dynamic routes to pre-render pages based on data.
- `getServerSideProps` (Server-side Rendering): Fetch data on **each request**.

In addition, we'll talk briefly about how to fetch data on the client side.

`getStaticProps` (Static Generation)

▶ **Version History**

If you export an `async` function called `getStaticProps` from a page, Next.js will pre-render this page at build time using the props returned by `getStaticProps`.

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

The `context` parameter is an object containing the following keys:

- `params` contains the route parameters for pages using dynamic routes. For example, if the page name is `[id].js` , then `params` will look like `{ id: ... }`. To learn more, take a look at the Dynamic Routing documentation. You should use this together with `getStaticPaths`, which we'll explain later.
- `preview` is `true` if the page is in the preview mode and `undefined` otherwise. See the Preview Mode documentation.
- `previewData` contains the preview data set by `setPreviewData`. See the Preview Mode documentation.
- `locale` contains the active locale (if enabled).
- `locales` contains all supported locales (if enabled).
- `defaultLocale` contains the configured default locale (if enabled).

`getStaticProps` should return an object with:

- `props` - A **required** object with the props that will be received by the page component. It should be a serializable object
- `revalidate` - An **optional** amount in seconds after which a page re-generation can occur. More on Incremental Static Regeneration
- `notFound` - An **optional** boolean value to allow the page to return a 404 status and page. Below is an example of how it works:

```
export async function getStaticProps(context) {
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  if (!data) {
    return {
      notFound: true,
    }
  }

  return {
    props: {}, // will be passed to the page component as props
  }
}
```

> **Note**: `notFound` is not needed for `fallback: false` mode as only paths returned from `getStaticPaths` will be pre-rendered.

- `redirect` - An **optional** redirect value to allow redirecting to internal and external resources. It should match the shape of `{ destination: string, permanent: boolean }`. In some rare cases, you might need to assign a custom status code for older HTTP Clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent` property, but not both. Below is an example of how it works:

```
export async function getStaticProps(context) {
  const res = await fetch(`https://...`)
  const data = await res.json()

  if (!data) {
    return {
      redirect: {
        destination: '/',
        permanent: false,
      },
    }
  }

  return {
    props: {}, // will be passed to the page component as props
  }
}
```

> **Note**: Redirecting at build-time is currently not allowed and if the redirects are known at build-time they should be added in `next.config.js`.

Note: You can import modules in top-level scope for use in `getStaticProps`. Imports used in `getStaticProps` will not be bundled for the client-side.

This means you can write **server-side code directly in `getStaticProps`**. This includes reading from the filesystem or a database.

Note: You should not use `fetch()` to call an API route in `getStaticProps`. Instead, directly import the logic used inside your API route. You may need to slightly refactor your code for this approach.

Fetching from an external API is fine!

Simple Example

Here's an example which uses `getStaticProps` to fetch a list of blog posts from a CMS (content management system). This example is also in the Pages documentation.

```
// posts will be populated at build time by getStaticProps()
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. See the "Technical details" section.
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
```

When should I use `getStaticProps`?

You should use `getStaticProps` if:

- The data required to render the page is available at build time ahead of a user's request.
- The data comes from a headless CMS.
- The data can be publicly cached (not user-specific).
- The page must be pre-rendered (for SEO) and be very fast — `getStaticProps` generates HTML and JSON files, both of which can be cached by a CDN for performance.

TypeScript: Use `GetStaticProps`

For TypeScript, you can use the `GetStaticProps` type from `next`:

```
import { GetStaticProps } from 'next'

export const getStaticProps: GetStaticProps = async (context) => {
  // ...
}
```

If you want to get inferred typings for your props, you can use `InferGetStaticPropsType<typeof getStaticProps>`, like this:

```
import { InferGetStaticPropsType } from 'next'

type Post = {
  author: string
  content: string
}

export const getStaticProps = async () => {
  const res = await fetch('https://.../posts')
  const posts: Post[] = await res.json()

  return {
    props: {
      posts,
    },
  }
}

function Blog({ posts }: InferGetStaticPropsType<typeof getStaticProps>) {
  // will resolve posts to type Post[]
}

export default Blog
```

Incremental Static Regeneration

This feature was introduced in Next.js 9.5 and up. If you're using older versions of Next.js, please upgrade before trying Incremental Static Regeneration.

▼ **Examples**

- Static Reactions Demo

With `getStaticProps` you don't have to stop relying on dynamic content, as **static content can also be dynamic**. Incremental Static Regeneration allows you to update existing pages by re-rendering them in the background as traffic comes in.

Inspired by stale-while-revalidate, background regeneration ensures traffic is served uninterruptedly, always from static storage, and the newly built page is pushed only after it's done generating.

Consider our previous `getStaticProps` example, but now with regeneration enabled:

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

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// revalidation is enabled and a new request comes in
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js will attempt to re-generate the page:
    // - When a request comes in
    // - At most once every second
    revalidate: 1, // In seconds
  }
}

export default Blog
```

Now the list of blog posts will be revalidated once per second; if you add a new blog post it will be available almost immediately, without having to re-build your app or make a new deployment.

This works perfectly with `fallback: true`. Because now you can have a list of posts that's always up to date with the latest posts, and have a blog post page that generates blog posts on-demand, no matter how many posts you add or update.

Static content at scale

Unlike traditional SSR, Incremental Static Regeneration ensures you retain the benefits of static:

- No spikes in latency. Pages are served consistently fast
- Pages never go offline. If the background page re-generation fails, the old page remains unaltered
- Low database and backend load. Pages are re-computed at most once concurrently

Reading files: Use `process.cwd()`

Files can be read directly from the filesystem in `getStaticProps`.

In order to do so you have to get the full path to a file.

Since Next.js compiles your code into a separate directory you can't use `` `__dirname` `` as the path it will return will be different from the pages directory.

Instead you can use `` `process.cwd()` `` which gives you the directory where Next.js is being executed.

```
import { promises as fs } from 'fs'
import path from 'path'

// posts will be populated at build time by getStaticProps()
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li>
          <h3>{post.filename}</h3>
          <p>{post.content}</p>
        </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. See the "Technical details" section.
export async function getStaticProps() {
  const postsDirectory = path.join(process.cwd(), 'posts')
  const filenames = await fs.readdir(postsDirectory)

  const posts = filenames.map(async (filename) => {
    const filePath = path.join(postsDirectory, filename)
    const fileContents = await fs.readFile(filePath, 'utf8')

    // Generally you would parse/transform the contents
    // For example you can transform markdown to HTML here
```

```
      return {
        filename,
        content: fileContents,
      }
    })
    // By returning { props: posts }, the Blog component
    // will receive `posts` as a prop at build time
    return {
      props: {
        posts: await Promise.all(posts),
      },
    }
  }

  export default Blog
```

Technical details

Only runs at build time

Because `getStaticProps` runs at build time, it does **not** receive data that's only available during request time, such as query parameters or HTTP headers as it generates static HTML.

Write server-side code directly

Note that `getStaticProps` runs only on the server-side. It will never be run on the client-side. It won't even be included in the JS bundle for the browser. That means you can write code such as direct database queries without them being sent to browsers. You should not fetch an **API route** from `getStaticProps` — instead, you can write the server-side code directly in `getStaticProps`.

You can use this tool to verify what Next.js eliminates from the client-side bundle.

Statically Generates both HTML and JSON

When a page with `getStaticProps` is pre-rendered at build time, in addition to the page HTML file, Next.js generates a JSON file holding the result of running `getStaticProps`.

This JSON file will be used in client-side routing through `next/link` (documentation) or `next/router` (documentation). When you navigate to a page that's pre-rendered using `getStaticProps`, Next.js fetches this JSON file (pre-computed at build time) and uses it as the props for the page component. This means that client-side page transitions will **not** call `getStaticProps` as only the exported JSON is used.

Only allowed in a page

`getStaticProps` can only be exported from a **page**. You can't export it from non-page files.

One of the reasons for this restriction is that React needs to have all the required data before the page is rendered.

Also, you must use `export async function getStaticProps() {}` — it will **not** work if you add `getStaticProps` as a property of the page component.

Runs on every request in development

In development (`next dev`), `getStaticProps` will be called on every request.

Preview Mode

In some cases, you might want to temporarily bypass Static Generation and render the page at **request time** instead of build time. For example, you might be using a headless CMS and want to preview drafts before they're published.

This use case is supported by Next.js by the feature called **Preview Mode**. Learn more on the Preview Mode documentation.

`getStaticPaths` (Static Generation)

▶ **Version History**

If a page has dynamic routes (documentation) and uses `getStaticProps` it needs to define a list of paths that have to be rendered to HTML at build time.

If you export an `async` function called `getStaticPaths` from a page that uses dynamic routes, Next.js will statically pre-render all the paths specified by `getStaticPaths`.

```
export async function getStaticPaths() {
  return {
    paths: [
      { params: { ... } } // See the "paths" section below
    ],
    fallback: true or false // See the "fallback" section below
  };
}
```

The `paths` key (required)

The `paths` key determines which paths will be pre-rendered. For example, suppose that you have a page that uses dynamic routes named `pages/posts/[id].js`. If you export `getStaticPaths` from this page and return the following for `paths`:

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

Then Next.js will statically generate `posts/1` and `posts/2` at build time using the page component in `pages/posts/[id].js`.

Note that the value for each `params` must match the parameters used in the page name:

- If the page name is `pages/posts/[postId]/[commentId]`, then `params` should contain `postId` and `commentId`.
- If the page name uses catch-all routes, for example `pages/[...slug]`, then `params` should contain `slug` which is an array. For example, if this array is `['foo', 'bar']`, then Next.js will statically generate the page at `/foo/bar`.
- If the page uses an optional catch-all route, supply `null`, `[]`, `undefined` or `false` to render the root-most route. For example, if you supply `slug: false` for `pages/[[...slug]]`, Next.js will statically generate the page `/`.

The `fallback` key (required)

The object returned by `getStaticPaths` must contain a boolean `fallback` key.

`fallback: false`

If `fallback` is `false`, then any paths not returned by `getStaticPaths` will result in a **404 page**. You can do this if you have a small number of paths to pre-render - so they are all statically generated during build time. It's also useful when the new pages are not added often. If you add more items to the data source and need to render the new pages, you'd need to run the build again.

Here's an example which pre-renders one blog post per page called `pages/posts/[id].js`. The list of blog posts will be fetched from a CMS and returned by `getStaticPaths` . Then, for each page, it fetches the post data from a CMS using `getStaticProps`. This example is also in the Pages documentation.

```
// 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
```

`fallback: true`

▶ **Examples**

If `fallback` is `true`, then the behavior of `getStaticProps` changes:

- The paths returned from `getStaticPaths` will be rendered to HTML at build time by `getStaticProps`.
- The paths that have not been generated at build time will **not** result in a 404 page. Instead, Next.js will serve a “fallback” version of the page on the first request to such a path (see “Fallback pages” below for details).
- In the background, Next.js will statically generate the requested path HTML and JSON. This includes running `getStaticProps`.
- When that’s done, the browser receives the JSON for the generated path. This will be used to automatically render the page with the required props. From the user’s perspective, the page will be swapped from the fallback page to the full page.
- At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, just like other pages pre-rendered at build time.

> `fallback: true` is not supported when using `next export`.

Fallback pages

In the “fallback” version of a page:

- The page’s props will be empty.

- Using the router, you can detect if the fallback is being rendered, `router.isFallback` will be `true`.

Here’s an example that uses `isFallback`:

```
// pages/posts/[id].js
import { useRouter } from 'next/router'

function Post({ post }) {
  const router = useRouter()

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

  // Render post...
}

// This function gets called at build time
export async function getStaticPaths() {
  return {
    // Only `/posts/1` and `/posts/2` are generated at build time
    paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
    // Enable statically generating additional pages
    // For example: `/posts/3`
    fallback: true,
  }
}

// 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 },
      // Re-generate the post at most once per second
      // if a request comes in
      revalidate: 1,
    }
  }

  export default Post
```

When is `fallback: true` useful?

`fallback: true` is useful if your app has a very large number of static pages that depend on data (think: a very large e-commerce site). You want to pre-render all product pages, but then your builds would take forever.

Instead, you may statically generate a small subset of pages and use `fallback: true` for the rest. When someone requests a page that's not generated yet, the user will see the page with a loading indicator. Shortly after, `getStaticProps` finishes and the page will be rendered with the requested data. From now on, everyone who requests the same page will get the statically pre-rendered page.

This ensures that users always have a fast experience while preserving fast builds and the benefits of Static Generation.

`fallback: true` will not update generated pages, for that take a look at Incremental Static Regeneration.

`fallback: 'blocking'`

If `fallback` is `'blocking'`, new paths not returned by `getStaticPaths` will wait for the HTML to be generated, identical to SSR (hence why blocking), and then be cached for future requests so it only happens once per path.

`getStaticProps` will behave as follows:

- The paths returned from `getStaticPaths` will be rendered to HTML at build time by `getStaticProps`.
- The paths that have not been generated at build time will **not** result in a 404 page. Instead, Next.js will SSR on the first request and return the generated HTML.
- When that's done, the browser receives the HTML for the generated path. From the user's perspective, it will transition from "the browser is requesting the page" to "the full page is loaded". There is no flash of loading/fallback state.
- At the same time, Next.js adds this path to the list of pre-rendered pages. Subsequent requests to the same path will serve the generated page, just like other pages pre-rendered at build time.

`fallback: 'blocking'` will not update generated pages by default. To update generated pages, use Incremental Static Regeneration in conjunction with `fallback: 'blocking'`.

> `fallback: 'blocking'` is not supported when using `next export`.

When should I use `getStaticPaths`?

You should use `getStaticPaths` if you're statically pre-rendering pages that use dynamic routes.

TypeScript: Use `GetStaticPaths`

For TypeScript, you can use the `GetStaticPaths` type from `next`:

```
import { GetStaticPaths } from 'next'
```

```
export const getStaticPaths: GetStaticPaths = async () => {
  // ...
}
```

Technical details

Use together with `getStaticProps`

When you use `getStaticProps` on a page with dynamic route parameters, you must use `getStaticPaths`.

You cannot use `getStaticPaths` with `getServerSideProps`.

Only runs at build time on server-side

`getStaticPaths` only runs at build time on server-side.

Only allowed in a page

`getStaticPaths` can only be exported from a **page**. You can't export it from non-page files.

Also, you must use `export async function getStaticPaths() {}` — it will **not** work if you add `getStaticPaths` as a property of the page component.

Runs on every request in development

In development (`next dev`), `getStaticPaths` will be called on every request.

`getServerSideProps` (Server-side Rendering)

▶ **Version History**

If you export an `async` function called `getServerSideProps` from a page, Next.js will pre-render this page on each request using the data returned by `getServerSideProps`.

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

The `context` parameter is an object containing the following keys:

- `params`: If this page uses a dynamic route, `params` contains the route parameters. If the page name is `[id].js` , then `params` will look like `{ id: ... }`. To learn more, take a look at the Dynamic Routing documentation.
- `req`: The HTTP IncomingMessage object.
- `res`: The HTTP response object.
- `query`: An object representing the query string.
- `preview`: `preview` is `true` if the page is in the preview mode and `false` otherwise. See the Preview Mode documentation.
- `previewData`: The preview data set by `setPreviewData`. See the Preview Mode documentation.
- `resolvedUrl`: A normalized version of the request URL that strips the `_next/data` prefix for client transitions and includes original query values.
- `locale` contains the active locale (if enabled).
- `locales` contains all supported locales (if enabled).

- `defaultLocale` contains the configured default locale (if enabled).

`getServerSideProps` should return an object with:

- `props` - A **required** object with the props that will be received by the page component. It should be a serializable object

- `notFound` - An **optional** boolean value to allow the page to return a 404 status and page. Below is an example of how it works:

```
export async function getServerSideProps(context) {
  const res = await fetch(`https://...`)
  const data = await res.json()

  if (!data) {
    return {
      notFound: true,
    }
  }

  return {
    props: {}, // will be passed to the page component as props
  }
}
```

- `redirect` - An **optional** redirect value to allow redirecting to internal and external resources. It should match the shape of `{ destination: string, permanent: boolean }`. In some rare cases, you might need to assign a custom status code for older HTTP Clients to properly redirect. In these cases, you can use the `statusCode` property instead of the `permanent` property, but not both. Below is an example of how it works:

```
export async function getServerSideProps(context) {
  const res = await fetch(`https://.../data`)
  const data = await res.json()

  if (!data) {
    return {
      redirect: {
        destination: '/',
        permanent: false,
      },
    }
  }

  return {
    props: {}, // will be passed to the page component as props
  }
}
```

Note: You can import modules in top-level scope for use in `getServerSideProps`. Imports used in `getServerSideProps` will not be bundled for the client-side.

This means you can write **server-side code directly in `getServerSideProps`**. This includes reading from the filesystem or a database.

Note: You should not use `fetch()` to call an API route in `getServerSideProps`. Instead, directly import the logic used inside your API route. You may need to slightly refactor your code for this approach.

Fetching from an external API is fine!

Simple example

Here's an example which uses `getServerSideProps` to fetch data at request time and pre-renders it. This example is also in the Pages documentation.

```
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
```

When should I use `getServerSideProps`?

You should use `getServerSideProps` only if you need to pre-render a page whose data must be fetched at request time. Time to first byte (TTFB) will be slower than `getStaticProps` because the server must compute the result on every request, and the result cannot be cached by a CDN without extra configuration.

If you don’t need to pre-render the data, then you should consider fetching data on the client side. Click here to learn more.

TypeScript: Use `GetServerSideProps`

For TypeScript, you can use the `GetServerSideProps` type from `next`:

```
import { GetServerSideProps } from 'next'

export const getServerSideProps: GetServerSideProps = async (context) => {
  // ...
}
```

If you want to get inferred typings for your props, you can use `` `InferGetServerSidePropsType<typeof getServerSideProps>` ``, like this:

```
import { InferGetServerSidePropsType } from 'next'

type Data = { ... }

export const getServerSideProps = async () => {
  const res = await fetch('https://.../data')
  const data: Data = await res.json()

  return {
    props: {
      data,
    },
  }
}

function Page({ data }: InferGetServerSidePropsType<typeof getServerSideProps
  // will resolve posts to type Data
}

export default Page
```

Technical details

Only runs on server-side

`` `getServerSideProps` `` only runs on server-side and never runs on the browser. If a page uses `` `getServerSideProps` ``, then:

- When you request this page directly, `getServerSideProps` runs at the request time, and this page will be pre-rendered with the returned props.
- When you request this page on client-side page transitions through `next/link` (documentation) or `next/router` (documentation), Next.js sends an API request to the server, which runs `getServerSideProps`. It'll return JSON that contains the result of running `getServerSideProps`, and the JSON will be used to render the page. All this work will be handled automatically by Next.js, so you don't need to do anything extra as long as you have `getServerSideProps` defined.

You can use this tool to verify what Next.js eliminates from the client-side bundle.

Only allowed in a page

`getServerSideProps` can only be exported from a **page**. You can't export it from non-page files.

Also, you must use `export async function getServerSideProps() {}` — it will **not** work if you add `getServerSideProps` as a property of the page component.

Fetching data on the client side

If your page contains frequently updating data, and you don't need to pre-render the data, you can fetch the data on the client side. An example of this is user-specific data. Here's how it works:

- First, immediately show the page without data. Parts of the page can be pre-rendered using Static Generation. You can show loading states for missing data.
- Then, fetch the data on the client side and display it when ready.

This approach works well for user dashboard pages, for example. Because a dashboard is a private, user-specific page, SEO is not relevant and the page doesn't need to be pre-rendered. The data is frequently updated, which requires request-time data fetching.

SWR

The team behind Next.js has created a React hook for data fetching called **SWR**. We highly recommend it if you're fetching data on the client side. It handles caching, revalidation, focus tracking, refetching on interval, and more. And you can use it like so:

```
import useSWR from 'swr'

function Profile() {
  const { data, error } = useSWR('/api/user', fetch)

  if (error) return <div>failed to load</div>
  if (!data) return <div>loading...</div>
  return <div>hello {data.name}!</div>
}
```

Check out the SWR documentation to learn more.

Learn more

We recommend you to read the following sections next:

Preview Mode:
Learn more about the preview mode in Next.js.

Routing:
Learn more about routing in Next.js.

TypeScript: Add TypeScript to your pages.

‹ Pages   Built-in CSS Support ›

Was this helpful?

😭 😕 😀 🤩

Edit this page on GitHub