# getStaticPaths

When exporting a function called `getStaticPaths` from a page that uses Dynamic Routes, Next.js will statically pre-render all the paths specified by `getStaticPaths`.

pages/repo/[name].tsx

```tsx
import type {
  InferGetStaticPropsType,
  GetStaticProps,
  GetStaticPaths,
} from 'next';

type Repo = {
  name: string;
  stargazers_count: number;
};

export const getStaticPaths: GetStaticPaths = async () => {
  return {
    paths: [
      {
        params: {
          name: 'next.js',
        },
      }, // See the "paths" section below
    ],
    fallback: true, // false or "blocking"
  };
};

export const getStaticProps: GetStaticProps<{
  repo: Repo;
}> = async () => {
  const res = await fetch('https://api.github.com/repos/vercel/next.js');
  const repo = await res.json();
  return { props: { repo } };
};

export default function Page({
  repo,
}: InferGetStaticPropsType<typeof getStaticProps>) {
```

```
  return repo.stargazers_count;
}
```

---

## getStaticPaths return values

The `getStaticPaths` function should return an object with the following **required** properties:

### `paths`

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' },
      // with i18n configured the locale for the path can be returned as well
      locale: "en",
    },
  ],
  fallback: ...
}
```

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

The value for each `params` object 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 like `pages/[...slug]`, then `params` should contain `slug` (which is an array). If this array is `['hello', 'world']`, then Next.js will statically generate the page at `/hello/world`.
- If the page uses an optional catch-all route, use `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 `params` strings are **case-sensitive** and ideally should be normalized to ensure the paths are generated correctly. For example, if `WoRLD` is returned for a param it will only match if `WoRLD` is the actual path visited, not `world` or `World`.

Separate of the `params` object a `locale` field can be returned when i18n is configured, which configures the locale for the path being generated.

### `fallback: false`

If `fallback` is `false`, then any paths not returned by `getStaticPaths` will result in a **404 page**.

When `next build` is run, Next.js will check if `getStaticPaths` returned `fallback: false`, it will then build **only** the paths returned by `getStaticPaths`. This option is useful if you have a small number of paths to create, or new page data is not added often. If you find that you need to add more paths, and you have `fallback: false`, you will need to run `next build` again so that the new paths can be generated.

The following example 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`.

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();
```

```
27
28    // Pass post data to the page via props
29    return { props: { post } };
30  }
31
32  export default Post;
```

## `fallback: true`

▶ **Examples**

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

- 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. Web crawlers, such as Google, won't be served a fallback and instead the path will behave as in `fallback: 'blocking'`.
- When a page with `fallback: true` is navigated to through `next/link` or `next/router` (client-side) Next.js will *not* serve a fallback and instead the page will behave as `fallback: 'blocking'`.
- In the background, Next.js will statically generate the requested path `HTML` and `JSON`. This includes running `getStaticProps`.
- When complete, 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, like other pages pre-rendered at build time.

**Note**: `fallback: true` is not supported when using `output: 'export'`.

### When is `fallback: true` useful?

`fallback: true` is useful if your app has a very large number of static pages that depend on data (such as a very large e-commerce site). If you want to pre-render all product pages, the builds would take a very long time.

Instead, you may statically generate a small subset of pages and use `fallback: true` for the rest. When someone requests a page that is not generated yet, the user will see the page with a loading indicator or skeleton component.

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 complete, 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, 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'`.

> **Note**: `fallback: 'blocking'` is not supported when using `output: '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`.

The following example showcases using `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;
```

## Version History

| Version | Changes |
| --- | --- |
| `v13.4.0` | App Router is now stable with simplifed data fetching, including `generateStaticParams()` |
| `v12.2.0` | On-Demand Incremental Static Regeneration is stable. |
| `v12.1.0` | On-Demand Incremental Static Regeneration added (beta). |

| Version | Changes |
| --- | --- |
| `v9.5.0` | Stable Incremental Static Regeneration |
| `v9.3.0` | `getStaticPaths` introduced. |