getStaticProps

Exporting a function called `getStaticProps` will pre-render a page at build time using the props returned from the function:

pages/index.tsx

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

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

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;
}
```

You can import modules in top-level scope for use in `getStaticProps`. Imports used will **not be bundled for the client-side**. This means you can write **server-side code directly in `getStaticProps`**, including fetching data from your database.

Context parameter

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

Name	Description
`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: ... }`. You should use this together with `getStaticPaths`, which we'll explain later.
`preview`	(Deprecated for `draftMode`) `preview` is `true` if the page is in the Preview Mode and `false` otherwise.
`previewData`	(Deprecated for `draftMode`) The preview data set by `setPreviewData`.
`draftMode`	`draftMode` is `true` if the page is in the Draft Mode and `false` otherwise.
`locale`	Contains the active locale (if enabled).
`locales`	Contains all supported locales (if enabled).
`defaultLocale`	Contains the configured default locale (if enabled).

getStaticProps return values

The `getStaticProps` function should return an object containing either `props`, `redirect`, or `notFound` followed by an **optional** `revalidate` property.

`props`

The `props` object is a key-value pair, where each value is received by the page component. It should be a serializable object↗ so that any props passed, could be serialized with `JSON.stringify`↗.

```
export async function getStaticProps(context) {
  return {
    props: { message: `Next.js is awesome` }, // will be passed to the page component as
  };
}
```

`revalidate`

The `revalidate` property is the amount in seconds after which a page re-generation can occur (defaults to `false` or no revalidation).

```
// 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 10 seconds
    revalidate: 10, // In seconds
  };
}
```

Learn more about Incremental Static Regeneration.

The cache status of a page leveraging ISR can be determined by reading the value of the `x-nextjs-cache` response header. The possible values are the following:

- `MISS` - the path is not in the cache (occurs at most once, on the first visit)
- `STALE` - the path is in the cache but exceeded the revalidate time so it will be updated in the background
- `HIT` - the path is in the cache and has not exceeded the revalidate time

`notFound`

The `notFound` boolean allows the page to return a `404` status and 404 Page. With `notFound: true`, the page will return a `404` even if there was a successfully generated page before. This is meant to support use cases like user-generated content getting removed by its author. Note, `notFound` follows the same `revalidate` behavior described here.

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

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

  return {
    props: { data }, // 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`

The `redirect` object allows redirecting to internal or 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**. You can also set `basePath: false` similar to redirects in `next.config.js`.

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

  if (!data) {
    return {
      redirect: {
        destination: '/',
        permanent: false,
        // statusCode: 301
      },
    };
  }

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

If the redirects are known at build-time, they should be added in `next.config.js` instead.

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 returns will be different from the Pages Router.

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

```jsx
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.
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;
```

Version History

Version	Changes
`v13.4.0`	App Router is now stable with simplifed data fetching
`v12.2.0`	On-Demand Incremental Static Regeneration is stable.
`v12.1.0`	On-Demand Incremental Static Regeneration added (beta).
`v10.0.0`	`locale`, `locales`, `defaultLocale`, and `notFound` options added.
`v10.0.0`	`fallback: 'blocking'` return option added.
`v9.5.0`	Stable Incremental Static Regeneration
`v9.3.0`	`getStaticProps` introduced.