

> Menu

Pages Router

Before Next.js 13, the Pages Router was the main way to create routes in Next.js. It used an intuitive file-system router to map each file to a route. The Pages Router is still supported in newer versions of Next.js, but we recommend migrating to the new [App Router](#) to leverage React's latest features.

Use this section of the documentation for existing applications that use the Pages Router.

[Building Your Application](#)

Learn how to use Next.js features to build your application.

[API Reference](#)

Next.js API Reference for the Pages Router.

> Menu

Pages Router > API Reference

API Reference

Components

API Reference for Next.js built-in components in the Pages Router.

Functions

API Reference for Functions and Hooks in Pages Router.

next.config.js Options

Learn about the options available in next.config.js for the Pages Router.

create-next-app

create-next-app

Next.js CLI

Next.js CLI

Edge Runtime

API Reference for the Edge Runtime.

> Menu

Pages Router > API Reference > Components

Components

Font

API Reference for the Font Module

<Head>

Add custom elements to the `head` of your page with the built-in Head component.

<Image>

Optimize Images in your Next.js Application using the built-in `next/image` Component.

<Image> (Legacy)

Backwards compatible Image Optimization with the Legacy Image component.

<Link>

API reference for the <Link> component.

<Script>

Optimize third-party scripts in your Next.js application using the built-in `next/script` Component.

> Menu

Pages Router > ... > Components > Font

Font Module

This API reference will help you understand how to use [next/font/google](#) and [next/font/local](#). For features and usage, please see the [Optimizing Fonts](#) page.

Font Function Arguments

For usage, review [Google Fonts](#) and [Local Fonts](#).

Key	font/google	font/local	Type	Required
<code>src</code>	×	✓	String or Array of Objects	Yes
<code>weight</code>	✓	✓	String or Array	Required/Optional
<code>style</code>	✓	✓	String or Array	-
<code>subsets</code>	✓	✗	Array of Strings	-
<code>axes</code>	✓	✗	Array of Strings	-
<code>display</code>	✓	✓	String	-
<code>preload</code>	✓	✓	Boolean	-
<code>fallback</code>	✓	✓	Array of Strings	-
<code>adjustFontFallback</code>	✓	✓	Boolean or String	-
<code>variable</code>	✓	✓	String	-
<code>declarations</code>	✗	✓	Array of Objects	-

src

The path of the font file as a string or an array of objects (with type

`Array<{path: string, weight?: string, style?: string}>`) relative to the directory where the font loader function is called.

Used in `next/font/local`

- Required

Examples:

- `src: './fonts/my-font.woff2'` where `my-font.woff2` is placed in a directory named `fonts` inside the `app` directory
- `src:[{path: './inter/Inter-Thin.ttf', weight: '100', },{path: './inter/Inter-Regular.ttf', weight: '400', },{path: './inter/Inter-Bold-Italic.ttf', weight: '700', style: 'italic', },]`
- if the font loader function is called in `app/page.tsx` using `src:'..../styles/fonts/my-font.ttf'`, then `my-font.ttf` is placed in `styles/fonts` at the root of the project

weight

The font `weight` ↗ with the following possibilities:

- A string with possible values of the weights available for the specific font or a range of values if it's a [variable](#) ↗ font
- An array of weight values if the font is not a [variable google font](#) ↗. It applies to `next/font/google` only.

Used in `next/font/google` and `next/font/local`

- Required if the font being used is [not variable](#) ↗

Examples:

- `weight: '400'`: A string for a single weight value - for the font [Inter](#) ↗, the possible values are `'100'`, `'200'`, `'300'`, `'400'`, `'500'`, `'600'`, `'700'`, `'800'`, `'900'` or `'variable'` where `'variable'` is the default)
- `weight: '100 900'`: A string for the range between `100` and `900` for a variable font
- `weight: ['100', '400', '900']`: An array of 3 possible values for a non variable font

style

The font `style` ↗ with the following possibilities:

- A string [value](#) ↗ with default value of `'normal'`
- An array of style values if the font is not a [variable google font](#) ↗. It applies to `next/font/google` only.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `style: 'italic'`: A string - it can be `normal` or `italic` for `next/font/google`
- `style: 'oblique'`: A string - it can take any value for `next/font/local` but is expected to come from [standard font styles ↗](#)
- `style: ['italic', 'normal']`: An array of 2 values for `next/font/google` - the values are from `normal` and `italic`

subsets

The font [subsets ↗](#) defined by an array of string values with the names of each subset you would like to be [preloaded](#). Fonts specified via `subsets` will have a link preload tag injected into the head when the `preload` option is true, which is the default.

Used in `next/font/google`

- Optional

Examples:

- `subsets: ['latin']`: An array with the subset `latin`

You can find a list of all subsets on the Google Fonts page for your font.

axes

Some variable fonts have extra `axes` that can be included. By default, only the font weight is included to keep the file size down. The possible values of `axes` depend on the specific font.

Used in `next/font/google`

- Optional

Examples:

- `axes: ['sln1']`: An array with value `sln1` for the `Inter` variable font which has `sln1` as additional `axes` as shown [here ↗](#). You can find the possible `axes` values for your font by using the filter on the [Google variable fonts page ↗](#) and looking for axes other than `wght`

display

The font `display` ↗ with possible string `values` ↗ of `'auto'`, `'block'`, `'swap'`, `'fallback'` or `'optional'` with default value of `'swap'`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `display: 'optional'`: A string assigned to the `optional` value

preload

A boolean value that specifies whether the font should be `preloaded` or not. The default is `true`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `preload: false`

fallback

The fallback font to use if the font cannot be loaded. An array of strings of fallback fonts with no default.

- Optional

Used in `next/font/google` and `next/font/local`

Examples:

- `fallback: ['system-ui', 'arial']`: An array setting the fallback fonts to `system-ui` or `arial`

adjustFontFallback

- For `next/font/google`: A boolean value that sets whether an automatic fallback font should be used to reduce [Cumulative Layout Shift](#) ↗. The default is `true`.
- For `next/font/local`: A string or boolean `false` value that sets whether an automatic fallback font should be used to reduce [Cumulative Layout Shift](#) ↗. The possible values are `'Arial'`, `'Times New Roman'` or `false`. The default is `'Arial'`.

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `adjustFontFallback: false`: for `next/font/google`
- `adjustFontFallback: 'Times New Roman'`: for `next/font/local`

variable

A string value to define the CSS variable name to be used if the style is applied with the [CSS variable method](#).

Used in `next/font/google` and `next/font/local`

- Optional

Examples:

- `variable: '--my-font'`: The CSS variable `--my-font` is declared

declarations

An array of font face [descriptor ↗](#) key-value pairs that define the generated `@font-face` further.

Used in `next/font/local`

- Optional

Examples:

- `declarations: [{ prop: 'ascent-override', value: '90%' }]`

Applying Styles

You can apply the font styles in three ways:

- `className`
- `style`

className

Returns a read-only CSS `className` for the loaded font to be passed to an HTML element.

```
<p className={inter.className}>Hello, Next.js!</p>
```

style

Returns a read-only CSS `style` object for the loaded font to be passed to an HTML element, including `style.fontFamily` to access the font family name and fallback fonts.

```
<p style={inter.style}>Hello World</p>
```

CSS Variables

If you would like to set your styles in an external style sheet and specify additional options there, use the CSS variable method.

In addition to importing the font, also import the CSS file where the CSS variable is defined and set the `variable` option of the font loader object as follows:

app/page.tsx

TypeScript ▾

```
1 import { Inter } from 'next/font/google';
2 import styles from '../styles/component.module.css';
3
4 const inter = Inter({
5   variable: '--font-inter',
6 });
```

To use the font, set the `className` of the parent container of the text you would like to style to the font loader's `variable` value and the `className` of the text to the `styles` property from the external CSS file.

app/page.tsx

TypeScript ▾

```
1 <main className={inter.variable}>
2   <p className={styles.text}>Hello World</p>
3 </main>
```

Define the `text` selector class in the `component.module.css` CSS file as follows:

styles/component.module.css

```
1 .text {  
2   font-family: var(--font-inter);  
3   font-weight: 200;  
4   font-style: italic;  
5 }
```

In the example above, the text `Hello World` is styled using the `Inter` font and the generated font fallback with `font-weight: 200` and `font-style: italic`.

Using a font definitions file

Every time you call the `localFont` or Google font function, that font will be hosted as one instance in your application. Therefore, if you need to use the same font in multiple places, you should load it in one place and import the related font object where you need it. This is done using a font definitions file.

For example, create a `fonts.ts` file in a `styles` folder at the root of your app directory.

Then, specify your font definitions as follows:

styles/fonts.ts

TypeScript ▾

```
1 import { Inter, Lora, Source_Sans_Pro } from 'next/font/google';  
2 import localFont from 'next/font/local';  
3  
4 // define your variable fonts  
5 const inter = Inter();  
6 const lora = Lora();  
7 // define 2 weights of a non-variable font  
8 const sourceCodePro400 = Source_Sans_Pro({ weight: '400' });  
9 const sourceCodePro700 = Source_Sans_Pro({ weight: '700' });  
10 // define a custom local font where GreatVibes-Regular.ttf is stored in the styles folder  
11 const greatVibes = localFont({ src: './GreatVibes-Regular.ttf' });  
12  
13 export { inter, lora, sourceCodePro400, sourceCodePro700, greatVibes };
```

You can now use these definitions in your code as follows:

app/page.tsx

TypeScript ▾

```
1 import { inter, lora, sourceCodePro700, greatVibes } from '../styles/fonts';
2
3 export default function Page() {
4   return (
5     <div>
6       <p className={inter.className}>Hello world using Inter font</p>
7       <p style={lora.style}>Hello world using Lora font</p>
8       <p className={sourceCodePro700.className}>
9         Hello world using Source_Sans_Pro font with weight 700
10      </p>
11      <p className={greatVibes.className}>My title in Great Vibes font</p>
12    </div>
13  );
14}
```

To make it easier to access the font definitions in your code, you can define a path alias in your `tsconfig.json` or `jsconfig.json` files as follows:

 tsconfig.json

```
1 {
2   "compilerOptions": {
3     "paths": {
4       "@/fonts": ["./styles/fonts"]
5     }
6   }
7 }
```

You can now import any font definition as follows:

 app/about/page.tsx

TypeScript ▾

```
import { greatVibes, sourceCodePro400 } from '@/fonts';
```

Version Changes

Version	Changes
v13.2.0	<code>@next/font</code> renamed to <code>next/font</code> . Installation no longer required.
v13.0.0	<code>@next/font</code> was added.

> Menu

Pages Router > ... > Components > <Head>

<Head>

► Examples

We expose a built-in component for appending elements to the `head` of the page:

```
1 import Head from 'next/head';
2
3 function IndexPage() {
4   return (
5     <div>
6       <Head>
7         <title>My page title</title>
8       </Head>
9         <p>Hello world!</p>
10      </div>
11    );
12  }
13
14 export default IndexPage;
```

To avoid duplicate tags in your `head` you can use the `key` property, which will make sure the tag is only rendered once, as in the following example:

```
1 import Head from 'next/head';
2
3 function IndexPage() {
4   return (
5     <div>
6       <Head>
7         <title>My page title</title>
8         <meta property="og:title" content="My page title" key="title" />
9       </Head>
10      <Head>
11        <meta property="og:title" content="My new title" key="title" />
12      </Head>
13      <p>Hello world!</p>
14    </div>
15  );
16 }
```

```
17  
18 export default IndexPage;
```

In this case only the second `<meta property="og:title" />` is rendered. `meta` tags with duplicate `key` attributes are automatically handled.

The contents of `head` get cleared upon unmounting the component, so make sure each page completely defines what it needs in `head`, without making assumptions about what other pages added.

`title`, `meta` or any other elements (e.g. `script`) need to be contained as **direct** children of the `Head` element, or wrapped into maximum one level of `<React.Fragment>` or arrays—otherwise the tags won't be correctly picked up on client-side navigations.

We recommend using `next/script` in your component instead of manually creating a `<script>` in `next/head`.

> Menu

Pages Router > ... > Components > <Image>

<Image>

► Examples

Note: If you are using a version of Next.js prior to 13, you'll want to use the [next/legacy/image](#) documentation since the component was renamed.

This API reference will help you understand how to use [props](#) and [configuration options](#) available for the Image Component. For features and usage, please see the [Image Component](#) page.

JS app/page.js



```
1 import Image from 'next/image';
2
3 export default function Page() {
4   return (
5     <Image
6       src="/profile.png"
7       width={500}
8       height={500}
9       alt="Picture of the author"
10    />
11  );
12}
```

Props

Here's a summary of the props available for the Image Component:

Prop	Example	Type	Required
src	src="/profile.png"	String	Yes

Prop	Example	Type	Required
width	width={500}	Integer (px)	Yes
height	height={500}	Integer (px)	Yes
alt	alt="Picture of the author"	String	Yes
loader	loader={imageLoader}	Function	-
fill	fill={true}	Boolean	-
sizes	sizes="(max-width: 768px) 100vw"	String	-
quality	quality={80}	Integer (1-100)	-
priority	priority={true}	Boolean	-
placeholder	placeholder="blur"	String	-
style	style={{objectFit: "contain"}}	Object	-
onLoadingComplete	onLoadingComplete={img => done()}	Function	-
onLoad	onLoad={event => done()}	Function	-
onError	onError(event => fail())	Function	-
loading	loading="lazy"	String	-
blurDataURL	blurDataURL="data:image/jpeg..."	String	-

Required Props

The Image Component requires the following properties: `src`, `width`, `height`, and `alt`.

JS app/page.js

```

1 import Image from 'next/image';
2
3 export default function Page() {
4   return (
5     <div>
6       <Image
7         src="/profile.png"
8         width={500}
9         height={500}

```

```
10         alt="Picture of the author"
11     />
12   </div>
13 );
14 }
```

src

Must be one of the following:

- A [statically imported](#) image file
- A path string. This can be either an absolute external URL, or an internal path depending on the [loader](#) prop.

When using an external URL, you must add it to [remotePatterns](#) in `next.config.js`.

width

The `width` property represents the *rendered* width in pixels, so it will affect how large the image appears.

Required, except for [statically imported images](#) or images with the [fill](#) property.

height

The `height` property represents the *rendered* height in pixels, so it will affect how large the image appears.

Required, except for [statically imported images](#) or images with the [fill](#) property.

alt

The `alt` property is used to describe the image for screen readers and search engines. It is also the fallback text if images have been disabled or an error occurs while loading the image.

It should contain text that could replace the image [without changing the meaning of the page](#). It is not meant to supplement the image and should not repeat information that is already provided in the captions above or below the image.

If the image is [purely decorative](#) or [not intended for the user](#), the `alt` property should be an empty string (`alt=""`).

[Learn more](#)

Optional Props

The `<Image />` component accepts a number of additional properties beyond those which are required. This section describes the most commonly-used properties of the Image component. Find details about more rarely-used properties in the [Advanced Props](#) section.

loader

A custom function used to resolve image URLs.

A `loader` is a function returning a URL string for the image, given the following parameters:

- `src`
- `width`
- `quality`

Here is an example of using a custom loader:

```
1 import Image from 'next/image';
2
3 const imageLoader = ({ src, width, quality }) => {
4   return `https://example.com/${src}?w=${width}&q=${quality || 75}`;
5 }
6
7 export default function Page() {
8   return (
9     <Image
10       loader={imageLoader}
11       src="me.png"
12       alt="Picture of the author"
13       width={500}
14       height={500}
15     />
16   );
17 }
```

Alternatively, you can use the `loaderFile` configuration in `next.config.js` to configure every instance of `next/image` in your application, without passing a prop.

fill

```
fill={true} // {true} | {false}
```

A boolean that causes the image to fill the parent element instead of setting `width` and `height`.

The parent element *must* assign `position: "relative"`, `position: "fixed"`, or `position: "absolute"` style.

By default, the `img` element will automatically be assigned the `position: "absolute"` style.

The default image fit behavior will stretch the image to fit the container. You may prefer to set `object-fit: "contain"` for an image which is letterboxed to fit the container and preserve aspect ratio.

Alternatively, `object-fit: "cover"` will cause the image to fill the entire container and be cropped to preserve aspect ratio. For this to look correct, the `overflow: "hidden"` style should be assigned to the parent element.

For more information, see also:

- [position ↗](#)
- [object-fit ↗](#)
- [object-position ↗](#)

sizes

A string that provides information about how wide the image will be at different breakpoints. The value of `sizes` will greatly affect performance for images using `fill` or which are styled to have a responsive size.

The `sizes` property serves two important purposes related to image performance:

- First, the value of `sizes` is used by the browser to determine which size of the image to download, from `next/image`'s automatically-generated source set. When the browser chooses, it does not yet know the size of the image on the page, so it selects an image that is the same size or larger than the viewport. The `sizes` property allows you to tell the browser that the image will actually be smaller than full screen. If you don't specify a `sizes` value in an image with the `fill` property, a default value of `100vw` (full screen width) is used.
- Second, the `sizes` property configures how `next/image` automatically generates an image source set. If no `sizes` value is present, a small source set is generated, suitable for a fixed-size image. If `sizes` is defined, a large source set is generated, suitable for a responsive image. If the `sizes` property includes sizes such as `50vw`, which represent a percentage of the viewport width, then the source set is trimmed to not include any values which are too small to ever be necessary.

For example, if you know your styling will cause an image to be full-width on mobile devices, in a 2-column layout on tablets, and a 3-column layout on desktop displays, you should include a `sizes` property such as the following:

```
1 import Image from 'next/image';
2
3 export default function Page() {
4   return (
5     <div className="grid-element">
6       <Image
7         fill
8         src="/example.png"
9         sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw, 33vw"
10      />
11    </div>
12  );
13}
```

This example `sizes` could have a dramatic effect on performance metrics. Without the `33vw` `sizes`, the image selected from the server would be 3 times as wide as it needs to be. Because file size is proportional to the square of the width, without `sizes` the user would download an image that's 9 times larger than necessary.

Learn more about `srcset` and `sizes`:

- [web.dev ↗](#)
- [mdn ↗](#)

quality

```
quality={75} // {number 1-100}
```

The quality of the optimized image, an integer between `1` and `100`, where `100` is the best quality and therefore largest file size. Defaults to `75`.

priority

```
priority={false} // {false} | {true}
```

When true, the image will be considered high priority and [preload ↗](#). Lazy loading is automatically disabled for images using `priority`.

You should use the `priority` property on any image detected as the [Largest Contentful Paint \(LCP\) ↗](#) element. It may be appropriate to have multiple priority images, as different images may be the LCP element for different viewport sizes.

Should only be used when the image is visible above the fold. Defaults to `false`.

placeholder

```
placeholder = 'empty'; // {empty} | {blur}
```

A placeholder to use while the image is loading. Possible values are `blur` or `empty`. Defaults to `empty`.

When `blur`, the `blurDataURL` property will be used as the placeholder. If `src` is an object from a [static import](#) and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated.

For dynamic images, you must provide the `blurDataURL` property. Solutions such as [Placeholder](#) can help with `base64` generation.

When `empty`, there will be no placeholder while the image is loading, only empty space.

Try it out:

- [Demo the `blur` placeholder](#)
- [Demo the shimmer effect with `blurDataURL` prop](#)
- [Demo the color effect with `blurDataURL` prop](#)

Advanced Props

In some cases, you may need more advanced usage. The `<Image />` component optionally accepts the following advanced properties.

style

Allows passing CSS styles to the underlying image element.

`JS` components/ProfileImage.js

```
1 const imageStyle = {
2   borderRadius: '50%',
3   border: '1px solid #fff',
4 };
5
6 export default function ProfileImage() {
```

```
7   return <Image src="..." style={imageStyle} />;  
8 }
```

Remember that the required width and height props can interact with your styling. If you use styling to modify an image's width, you should also style its height to `auto` to preserve its intrinsic aspect ratio, or your image will be distorted.

onLoadingComplete

```
<Image onLoadingComplete={(img) => console.log(img.naturalWidth)} />
```

A callback function that is invoked once the image is completely loaded and the `placeholder` has been removed.

The callback function will be called with one argument, a reference to the underlying `` element.

onLoad

```
<Image onLoad={(e) => console.log(e.target.naturalWidth)} />
```

A callback function that is invoked when the image is loaded.

Note that the load event might occur before the placeholder is removed and the image is fully decoded.

Instead, use `onLoadingComplete`.

onError

```
<Image onError={(e) => console.error(e.target.id)} />
```

A callback function that is invoked if the image fails to load.

loading

Recommendation: This property is only meant for advanced use cases. Switching an image to load with `eager` will normally **hurt performance**. We recommend using the `priority` property instead, which will eagerly preload the image.

```
loading = 'lazy'; // {lazy} | {eager}
```

The loading behavior of the image. Defaults to `lazy`.

When `lazy`, defer loading the image until it reaches a calculated distance from the viewport.

When `eager`, load the image immediately.

Learn more about the [loading attribute ↗](#).

blurDataURL

A [Data URL ↗](#) to be used as a placeholder image before the `src` image successfully loads. Only takes effect when combined with `placeholder="blur"`.

Must be a base64-encoded image. It will be enlarged and blurred, so a very small image (10px or less) is recommended. Including larger images as placeholders may harm your application performance.

Try it out:

- [Demo the default `blurDataURL` prop ↗](#)
- [Demo the shimmer effect with `blurDataURL` prop ↗](#)
- [Demo the color effect with `blurDataURL` prop ↗](#)

You can also [generate a solid color Data URL ↗](#) to match the image.

unoptimized

```
unoptimized = {false} // {false} | {true}
```

When true, the source image will be served as-is instead of changing quality, size, or format. Defaults to `false`.

```
1 import Image from 'next/image';
2
3 const UnoptimizedImage = (props) => {
4   return <Image {...props} unoptimized />;
5 }
```

Since Next.js 12.3.0, this prop can be assigned to all images by updating `next.config.js` with the following configuration:

```
1 module.exports = {  
2   images: {  
3     unoptimized: true,  
4   },  
5 };
```

Other Props

Other properties on the `<Image />` component will be passed to the underlying `img` element with the exception of the following:

- `srcSet`. Use [Device Sizes](#) instead.
- `decoding`. It is always `"async"`.

Configuration Options

In addition to props, you can configure the Image Component in `next.config.js`. The following options are available:

remotePatterns

To protect your application from malicious users, configuration is required in order to use external images. This ensures that only external images from your account can be served from the Next.js Image Optimization API. These external images can be configured with the `remotePatterns` property in your `next.config.js` file, as shown below:

```
1 module.exports = {
2   images: {
3     remotePatterns: [
4       {
5         protocol: 'https',
6         hostname: 'example.com',
7         port: '',
8         pathname: '/account123/**',
9       },
10    ],
11  },
12};
```

Note: The example above will ensure the `src` property of `next/image` must start with `https://example.com/account123/`. Any other protocol, hostname, port, or unmatched path will respond with 400 Bad Request.

Below is another example of the `remotePatterns` property in the `next.config.js` file:

js next.config.js

```
1 module.exports = {
2   images: {
3     remotePatterns: [
4       {
5         protocol: 'https',
6         hostname: '**.example.com',
7       },
8     ],
9   },
10};
```

Note: The example above will ensure the `src` property of `next/image` must start with `https://img1.example.com` or `https://me.avatar.example.com` or any number of subdomains. Any other protocol or unmatched hostname will respond with 400 Bad Request.

Wildcard patterns can be used for both `pathname` and `hostname` and have the following syntax:

- `*` match a single path segment or subdomain
- `**` match any number of path segments at the end or subdomains at the beginning

The `**` syntax does not work in the middle of the pattern.

domains

Warning: We recommend configuring strict `remotePatterns` instead of `domains` in order to protect your application from malicious users. Only use `domains` if you own all the content served from the domain.

Similar to `remotePatterns`, the `domains` configuration can be used to provide a list of allowed hostnames for external images.

However, the `domains` configuration does not support wildcard pattern matching and it cannot restrict protocol, port, or pathname.

Below is an example of the `domains` property in the `next.config.js` file:

JS next.config.js

```
1 module.exports = {
2   images: {
3     domains: ['assets.acme.com'],
4   },
5 };
```

loaderFile

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure the `loaderFile` in your `next.config.js` like the following:

JS next.config.js

```
1 module.exports = {
2   images: {
3     loader: 'custom',
4     loaderFile: './my/image/loader.js',
5   },
6 };
```

This must point to a file relative to the root of your Next.js application. The file must export a default function that returns a string, for example:

```
1 export default function myImageLoader({ src, width, quality }) {
2   return `https://example.com/${src}?w=${width}&q=${quality || 75}`;
3 }
```

Alternatively, you can use the `loader` prop to configure each instance of `next/image`.

Examples:

Advanced

The following configuration is for advanced use cases and is usually not necessary. If you choose to configure the properties below, you will override any changes to the Next.js defaults in future updates.

deviceSizes

If you know the expected device widths of your users, you can specify a list of device width breakpoints using the `deviceSizes` property in `next.config.js`. These widths are used when the `next/image` component uses `sizes` prop to ensure the correct image is served for user's device.

If no configuration is provided, the default below is used.

JS next.config.js

```
1 module.exports = {
2   images: {
3     deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
4   },
5 }
```

imageSizes

You can specify a list of image widths using the `images.imageSizes` property in your `next.config.js` file. These widths are concatenated with the array of `device sizes` to form the full array of sizes used to generate image `srcset`s.

The reason there are two separate lists is that `imageSizes` is only used for images which provide a `sizes` prop, which indicates that the image is less than the full width of the screen. **Therefore, the sizes in `imageSizes` should all be smaller than the smallest size in `deviceSizes`.**

If no configuration is provided, the default below is used.

JS next.config.js

```
1 module.exports = {
2   images: {
3     imageSizes: [16, 32, 48, 64, 96, 128, 256, 384],
4   },
5 }
```

```
5 };
```

formats

The default [Image Optimization API](#) will automatically detect the browser's supported image formats via the request's `Accept` header.

If the `Accept` head matches more than one of the configured formats, the first match in the array is used. Therefore, the array order matters. If there is no match (or the source image is [animated](#)), the Image Optimization API will fallback to the original image's format.

If no configuration is provided, the default below is used.

js next.config.js

```
1 module.exports = {
2   images: {
3     formats: ['image/webp'],
4   },
5 };
```

You can enable AVIF support with the following configuration.

js next.config.js

```
1 module.exports = {
2   images: {
3     formats: ['image/avif', 'image/webp'],
4   },
5 };
```

Note: AVIF generally takes 20% longer to encode but it compresses 20% smaller compared to WebP. This means that the first time an image is requested, it will typically be slower and then subsequent requests that are cached will be faster.

Note: If you self-host with a Proxy/CDN in front of Next.js, you must configure the Proxy to forward the `Accept` header.

Caching Behavior

The following describes the caching algorithm for the default [loader](#). For all other loaders, please refer to your cloud provider's documentation.

Images are optimized dynamically upon request and stored in the `<distDir>/cache/images` directory. The optimized image file will be served for subsequent requests until the expiration is reached. When a request is made that matches a cached but expired file, the expired image is served stale immediately. Then the image is optimized again in the background (also called revalidation) and saved to the cache with the new expiration date.

The cache status of an image 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

The expiration (or rather Max Age) is defined by either the `minimumCacheTTL` configuration or the upstream image `Cache-Control` header, whichever is larger. Specifically, the `max-age` value of the `Cache-Control` header is used. If both `s-maxage` and `max-age` are found, then `s-maxage` is preferred. The `max-age` is also passed-through to any downstream clients including CDNs and browsers.

- You can configure `minimumCacheTTL` to increase the cache duration when the upstream image does not include `Cache-Control` header or the value is very low.
- You can configure `deviceSizes` and `imageSizes` to reduce the total number of possible generated images.
- You can configure `formats` to disable multiple formats in favor of a single image format.

`minimumCacheTTL`

You can configure the Time to Live (TTL) in seconds for cached optimized images. In many cases, it's better to use a [Static Image Import](#) which will automatically hash the file contents and cache the image forever with a `Cache-Control` header of `immutable`.

Js next.config.js

```
1 module.exports = {
2   images: {
3     minimumCacheTTL: 60,
4   },
5 };
```

The expiration (or rather Max Age) of the optimized image is defined by either the `minimumCacheTTL` or the upstream image `Cache-Control` header, whichever is larger.

If you need to change the caching behavior per image, you can configure `headers` to set the `Cache-Control` header on the upstream image (e.g. `/some-asset.jpg`, not `/_next/image` itself).

There is no mechanism to invalidate the cache at this time, so its best to keep `minimumCacheTTL` low. Otherwise you may need to manually change the `src` prop or delete `<distDir>/cache/images`.

disableStaticImages

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property.

In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

You can disable static image imports inside your `next.config.js`:

`JS` `next.config.js`

```
1 module.exports = {
2   images: {
3     disableStaticImages: true,
4   },
5 };
```

dangerouslyAllowSVG

The default `loader` does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper [Content Security Policy \(CSP\) headers](#).

If you need to serve SVG images with the default Image Optimization API, you can set `dangerouslyAllowSVG` inside your `next.config.js`:

`JS` `next.config.js`

```
1 module.exports = {
2   images: {
3     dangerouslyAllowSVG: true,
4     contentDispositionType: 'attachment',
5     contentSecurityPolicy: "default-src 'self'; script-src 'none'; sandbox;",
6   },
}
```

```
7 };
```

In addition, it is strongly recommended to also set `contentDispositionType` to force the browser to download the image, as well as `contentSecurityPolicy` to prevent scripts embedded in the image from executing.

Animated Images

The default `loader` will automatically bypass Image Optimization for animated images and serve the image as-is.

Auto-detection for animated files is best-effort and supports GIF, APNG, and WebP. If you want to explicitly bypass Image Optimization for a given animated image, use the `unoptimized` prop.

Known Browser Bugs

This `next/image` component uses browser native [lazy loading](#), which may fallback to eager loading for older browsers before Safari 15.4. When using the blur-up placeholder, older browsers before Safari 12 will fallback to empty placeholder. When using styles with `width / height` of `auto`, it is possible to cause [Layout Shift](#) on older browsers before Safari 15 that don't [preserve the aspect ratio](#). For more details, see [this MDN video](#).

- [Safari 15 and 16](#) display a gray border while loading. Safari 16.4 [fixed this issue](#). Possible solutions:
 - Use CSS `@supports (font: -apple-system-body) and (-webkit-appearance: none) {`
`img[loading="lazy"] { clip-path: inset(0.6px) } }`
 - Use `priority` if the image is above the fold
- [Firefox 67+](#) displays a white background while loading. Possible solutions:
 - Enable [AVIF formats](#)
 - Use `placeholder="blur"`

Version History

Version Changes

v13.2.0	<code>contentDispositionType</code> configuration added.
v13.0.6	<code>ref</code> prop added.
v13.0.0	The <code>next/image</code> import was renamed to <code>next/legacy/image</code> . The <code>next/future/image</code> import was renamed to <code>next/image</code> . A codemod is available to safely and automatically rename your imports. <code></code> wrapper removed. <code>layout</code> , <code>objectFit</code> , <code>objectPosition</code> , <code>lazyBoundary</code> , <code>lazyRoot</code> props removed. <code>alt</code> is required. <code>onLoadingComplete</code> receives reference to <code>img</code> element. Built-in loader config removed.
v12.3.0	<code>remotePatterns</code> and <code>unoptimized</code> configuration is stable.
v12.2.0	Experimental <code>remotePatterns</code> and experimental <code>unoptimized</code> configuration added. <code>layout="raw"</code> removed.
v12.1.1	<code>style</code> prop added. Experimental support for <code>layout="raw"</code> added.
v12.1.0	<code>dangerouslyAllowSVG</code> and <code>contentSecurityPolicy</code> configuration added.
v12.0.9	<code>lazyRoot</code> prop added.
v12.0.0	<code>formats</code> configuration added. AVIF support added. Wrapper <code><div></code> changed to <code></code> .
v11.1.0	<code>onLoadingComplete</code> and <code>lazyBoundary</code> props added.
v11.0.0	<code>src</code> prop support for static import. <code>placeholder</code> prop added. <code>blurDataURL</code> prop added.
v10.0.5	<code>loader</code> prop added.
v10.0.1	<code>layout</code> prop added.
v10.0.0	<code>next/image</code> introduced.

> Menu

Pages Router > ... > Components > <Image> (Legacy)

<Image> (Legacy)

► Examples

Starting with Next.js 13, the `next/image` component was rewritten to improve both the performance and developer experience. In order to provide a backwards compatible upgrade solution, the old `next/image` was renamed to `next/legacy/image`.

View the **new** [next/image API Reference](#)

Comparison

Compared to `next/legacy/image`, the new `next/image` component has the following changes:

- Removes `` wrapper around `` in favor of [native computed aspect ratio ↗](#)
- Adds support for canonical `style` prop
 - Removes `layout` prop in favor of `style` or `className`
 - Removes `objectFit` prop in favor of `style` or `className`
 - Removes `objectPosition` prop in favor of `style` or `className`
- Removes `IntersectionObserver` implementation in favor of [native lazy loading ↗](#)
 - Removes `lazyBoundary` prop since there is no native equivalent
 - Removes `lazyRoot` prop since there is no native equivalent
- Removes `loader` config in favor of `Loader` prop
- Changed `alt` prop from optional to required
- Changed `onLoadingComplete` callback to receive reference to `` element

Required Props

The `<Image />` component requires the following properties.

src

Must be one of the following:

- A [statically imported](#) image file
- A path string. This can be either an absolute external URL, or an internal path depending on the [loader](#) prop or [loader configuration](#).

When using an external URL, you must add it to [remotePatterns](#) in `next.config.js`.

width

The `width` property can represent either the *rendered* width or *original* width in pixels, depending on the [layout](#) and [sizes](#) properties.

When using `layout="intrinsic"` or `layout="fixed"` the `width` property represents the *rendered* width in pixels, so it will affect how large the image appears.

When using `layout="responsive"`, `layout="fill"`, the `width` property represents the *original* width in pixels, so it will only affect the aspect ratio.

The `width` property is required, except for [statically imported images](#), or those with `layout="fill"`.

height

The `height` property can represent either the *rendered* height or *original* height in pixels, depending on the [layout](#) and [sizes](#) properties.

When using `layout="intrinsic"` or `layout="fixed"` the `height` property represents the *rendered* height in pixels, so it will affect how large the image appears.

When using `layout="responsive"`, `layout="fill"`, the `height` property represents the *original* height in pixels, so it will only affect the aspect ratio.

The `height` property is required, except for [statically imported images](#), or those with `layout="fill"`.

Optional Props

The `<Image />` component accepts a number of additional properties beyond those which are required. This section describes the most commonly-used properties of the Image component. Find details about more rarely-used properties in the [Advanced Props](#) section.

layout

The layout behavior of the image as the viewport changes size.

layout	Behavior	srcSet	sizes	Has wrapper and sizer
intrinsic (default)	Scale down to fit width of container, up to image size	<code>1x</code> , <code>2x</code> (based on imageSizes)	N/A	yes
fixed	Sized to <code>width</code> and <code>height</code> exactly	<code>1x</code> , <code>2x</code> (based on imageSizes)	N/A	yes
responsive	Scale to fit width of container	<code>640w</code> , <code>750w</code> , ... <code>2048w</code> , <code>3840w</code> (based on imageSizes and deviceSizes)	<code>100vw</code>	yes
fill	Grow in both X and Y axes to fill container	<code>640w</code> , <code>750w</code> , ... <code>2048w</code> , <code>3840w</code> (based on imageSizes and deviceSizes)	<code>100vw</code>	yes

- [Demo the `intrinsic` layout \(default\) ↗](#)

- When `intrinsic`, the image will scale the dimensions down for smaller viewports, but maintain the original dimensions for larger viewports.

- [Demo the `fixed` layout ↗](#)

- When `fixed`, the image dimensions will not change as the viewport changes (no responsiveness) similar to the native `img` element.

- [Demo the `responsive` layout ↗](#)

- When `responsive`, the image will scale the dimensions down for smaller viewports and scale up for larger viewports.

- Ensure the parent element uses `display: block` in their stylesheet.

- [Demo the `fill` layout ↗](#)

- When `fill`, the image will stretch both width and height to the dimensions of the parent element, provided the parent element is relative.

- This is usually paired with the `objectFit` property.
- Ensure the parent element has `position: relative` in their stylesheet.
- [Demo background image ↗](#)

loader

A custom function used to resolve URLs. Setting the loader as a prop on the Image component overrides the default loader defined in the [images section of `next.config.js`](#).

A `loader` is a function returning a URL string for the image, given the following parameters:

- `src`
- `width`
- `quality`

Here is an example of using a custom loader:

```
1 import Image from 'next/legacy/image';
2
3 const myLoader = ({ src, width, quality }) => {
4   return `https://example.com/${src}?w=${width}&q=${quality || 75}`;
5 }
6
7 const MyImage = (props) => {
8   return (
9     <Image
10       loader={myLoader}
11       src="me.png"
12       alt="Picture of the author"
13       width={500}
14       height={500}
15     />
16   );
17 }
```

sizes

A string that provides information about how wide the image will be at different breakpoints. The value of `sizes` will greatly affect performance for images using `layout="responsive"` or `layout="fill"`. It will be ignored for images using `layout="intrinsic"` or `layout="fixed"`.

The `sizes` property serves two important purposes related to image performance:

First, the value of `sizes` is used by the browser to determine which size of the image to download, from `next/legacy/image`'s automatically-generated source set. When the browser chooses, it does not yet know the size of the image on the page, so it selects an image that is the same size or larger than the viewport. The `sizes` property allows you to tell the browser that the image will actually be smaller than full screen. If you don't specify a `sizes` value, a default value of `100vw` (full screen width) is used.

Second, the `sizes` value is parsed and used to trim the values in the automatically-created source set. If the `sizes` property includes sizes such as `50vw`, which represent a percentage of the viewport width, then the source set is trimmed to not include any values which are too small to ever be necessary.

For example, if you know your styling will cause an image to be full-width on mobile devices, in a 2-column layout on tablets, and a 3-column layout on desktop displays, you should include a `sizes` property such as the following:

```
1 import Image from 'next/legacy/image';
2 const Example = () => (
3   <div className="grid-element">
4     <Image
5       src="/example.png"
6       layout="fill"
7       sizes="(max-width: 768px) 100vw,
8             (max-width: 1200px) 50vw,
9             33vw"
10    />
11  </div>
12 );
```

This example `sizes` could have a dramatic effect on performance metrics. Without the `33vw` `sizes`, the image selected from the server would be 3 times as wide as it needs to be. Because file size is proportional to the square of the width, without `sizes` the user would download an image that's 9 times larger than necessary.

Learn more about `srcset` and `sizes`:

- [web.dev ↗](#)
- [mdn ↗](#)

quality

The quality of the optimized image, an integer between `1` and `100` where `100` is the best quality. Defaults to `75`.

priority

When true, the image will be considered high priority and [preload ↗](#). Lazy loading is automatically disabled for images using `priority`.

You should use the `priority` property on any image detected as the [Largest Contentful Paint \(LCP\) ↗](#) element. It may be appropriate to have multiple priority images, as different images may be the LCP element for different viewport sizes.

Should only be used when the image is visible above the fold. Defaults to `false`.

placeholder

A placeholder to use while the image is loading. Possible values are `blur` or `empty`. Defaults to `empty`.

When `blur`, the `blurDataURL` property will be used as the placeholder. If `src` is an object from a [static import](#) and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated.

For dynamic images, you must provide the `blurDataURL` property. Solutions such as [Plaiceholder ↗](#) can help with `base64` generation.

When `empty`, there will be no placeholder while the image is loading, only empty space.

Try it out:

- [Demo the `blur` placeholder ↗](#)
 - [Demo the shimmer effect with `blurDataURL` prop ↗](#)
 - [Demo the color effect with `blurDataURL` prop ↗](#)
-

Advanced Props

In some cases, you may need more advanced usage. The `<Image />` component optionally accepts the following advanced properties.

style

Allows [passing CSS styles ↗](#) to the underlying image element.

Note that all `layout` modes apply their own styles to the image element, and these automatic styles take precedence over the `style` prop.

Also keep in mind that the required `width` and `height` props can interact with your styling. If you use styling to modify an image's `width`, you must set the `height="auto"` style as well, or your image will be distorted.

objectFit

Defines how the image will fit into its parent container when using `layout="fill"`.

This value is passed to the [object-fit CSS property ↗](#) for the `src` image.

objectPosition

Defines how the image is positioned within its parent element when using `layout="fill"`.

This value is passed to the [object-position CSS property ↗](#) applied to the image.

onLoadingComplete

A callback function that is invoked once the image is completely loaded and the [placeholder](#) has been removed.

The `onLoadingComplete` function accepts one parameter, an object with the following properties:

- [naturalWidth](#) ↗
- [naturalHeight](#) ↗

loading

Attention: This property is only meant for advanced usage. Switching an image to load with `eager` will normally **hurt performance**.

We recommend using the [priority](#) property instead, which properly loads the image eagerly for nearly all use cases.

The loading behavior of the image. Defaults to `lazy`.

When `lazy`, defer loading the image until it reaches a calculated distance from the viewport.

When `eager`, load the image immediately.

[Learn more ↗](#)

blurDataURL

A [Data URL ↗](#) to be used as a placeholder image before the `src` image successfully loads. Only takes effect when combined with `placeholder="blur"`.

Must be a base64-encoded image. It will be enlarged and blurred, so a very small image (10px or less) is recommended. Including larger images as placeholders may harm your application performance.

Try it out:

- [Demo the default `blurDataURL` prop ↗](#)
- [Demo the shimmer effect with `blurDataURL` prop ↗](#)
- [Demo the color effect with `blurDataURL` prop ↗](#)

You can also [generate a solid color Data URL ↗](#) to match the image.

lazyBoundary

A string (with similar syntax to the margin property) that acts as the bounding box used to detect the intersection of the viewport with the image and trigger lazy [loading](#). Defaults to `"200px"`.

If the image is nested in a scrollable parent element other than the root document, you will also need to assign the [lazyRoot](#) prop.

[Learn more ↗](#)

lazyRoot

A React [Ref ↗](#) pointing to the scrollable parent element. Defaults to `null` (the document viewport).

The Ref must point to a DOM element or a React component that [forwards the Ref ↗](#) to the underlying DOM element.

Example pointing to a DOM element

```
1 import Image from 'next/legacy/image';
2 import React from 'react';
3
4 const Example = () => {
5   const lazyRoot = React.useRef(null);
6
7   return (
8     <div ref={lazyRoot} style={{ overflowX: 'scroll', width: '500px' }}>
9       <Image lazyRoot={lazyRoot} src="/one.jpg" width="500" height="500" />
10      <Image lazyRoot={lazyRoot} src="/two.jpg" width="500" height="500" />
11    </div>
12 );
```

```
13 };
```

Example pointing to a React component

```
1 import Image from 'next/legacy/image';
2 import React from 'react';
3
4 const Container = React.forwardRef((props, ref) => {
5   return (
6     <div ref={ref} style={{ overflowX: 'scroll', width: '500px' }}>
7       {props.children}
8     </div>
9   );
10 });
11
12 const Example = () => {
13   const lazyRoot = React.useRef(null);
14
15   return (
16     <Container ref={lazyRoot}>
17       <Image lazyRoot={lazyRoot} src="/one.jpg" width="500" height="500" />
18       <Image lazyRoot={lazyRoot} src="/two.jpg" width="500" height="500" />
19     </Container>
20   );
21};
```

[Learn more ↗](#)

unoptimized

When true, the source image will be served as-is instead of changing quality, size, or format. Defaults to `false`.

```
1 import Image from 'next/image';
2
3 const UnoptimizedImage = (props) => {
4   return <Image {...props} unoptimized />;
5};
```

Since Next.js 12.3.0, this prop can be assigned to all images by updating `next.config.js` with the following configuration:

 `next.config.js`



```
1 module.exports = {
2   images: {
3     unoptimized: true,
```

```
4     },
5 }
```

Other Props

Other properties on the `<Image />` component will be passed to the underlying `img` element with the exception of the following:

- `srcSet`. Use [Device Sizes](#) instead.
- `ref`. Use [onLoadingComplete](#) instead.
- `decoding`. It is always `"async"`.

Configuration Options

Remote Patterns

To protect your application from malicious users, configuration is required in order to use external images. This ensures that only external images from your account can be served from the Next.js Image Optimization API. These external images can be configured with the `remotePatterns` property in your `next.config.js` file, as shown below:

`JS next.config.js`

```
1 module.exports = {
2   images: {
3     remotePatterns: [
4       {
5         protocol: 'https',
6         hostname: 'example.com',
7         port: '',
8         pathname: '/account123/**',
9       },
10     ],
11   },
12 };
```

Note: The example above will ensure the `src` property of `next/legacy/image` must start with `https://example.com/account123/`. Any other protocol, hostname, port, or unmatched path will respond with 400

Below is another example of the `remotePatterns` property in the `next.config.js` file:

JS next.config.js

```
1 module.exports = {
2   images: {
3     remotePatterns: [
4       {
5         protocol: 'https',
6         hostname: '**.example.com',
7       },
8     ],
9   },
10};
```

Note: The example above will ensure the `src` property of `next/legacy/image` must start with `https://img1.example.com` or `https://me.avatar.example.com` or any number of subdomains. Any other protocol or unmatched hostname will respond with 400 Bad Request.

Wildcard patterns can be used for both `pathname` and `hostname` and have the following syntax:

- `*` match a single path segment or subdomain
- `**` match any number of path segments at the end or subdomains at the beginning

The `**` syntax does not work in the middle of the pattern.

Domains

Warning: We recommend configuring strict `remotePatterns` instead of `domains` in order to protect your application from malicious users. Only use `domains` if you own all the content served from the domain.

Similar to `remotePatterns`, the `domains` configuration can be used to provide a list of allowed hostnames for external images.

However, the `domains` configuration does not support wildcard pattern matching and it cannot restrict protocol, port, or pathname.

Below is an example of the `domains` property in the `next.config.js` file:

JS next.config.js

```
1 module.exports = {
2   images: {
3     domains: ['assets.acme.com'],
4   },
5 };
```

Loader Configuration

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure the `loader` and `path` prefix in your `next.config.js` file. This allows you to use relative URLs for the Image `src` and automatically generate the correct absolute URL for your provider.

js next.config.js

```
1 module.exports = {
2   images: {
3     loader: 'imgix',
4     path: 'https://example.com/myaccount/',
5   },
6 };
```

Built-in Loaders

The following Image Optimization cloud providers are included:

- Default: Works automatically with `next dev`, `next start`, or a custom server
- [Vercel ↗](#): Works automatically when you deploy on Vercel, no configuration necessary. [Learn more ↗](#)
- [Imgix ↗](#): `loader: 'imgix'`
- [Cloudinary ↗](#): `loader: 'cloudinary'`
- [Akamai ↗](#): `loader: 'akamai'`
- Custom: `loader: 'custom'` use a custom cloud provider by implementing the `loader` prop on the `next/legacy/image` component

If you need a different provider, you can use the `loader` prop with `next/legacy/image`.

Images can not be optimized at build time using `output: 'export'`, only on-demand. To use `next/legacy/image` with `output: 'export'`, you will need to use a different loader than the default. [Read more in the discussion. ↗](#)

The `next/legacy/image` component's default loader uses [squoosh](#) ↗ because it is quick to install and suitable for a development environment. When using `next start` in your production environment, it is strongly recommended that you install [sharp](#) ↗ by running `yarn add sharp` in your project directory. This is not necessary for Vercel deployments, as `sharp` is installed automatically.

Advanced

The following configuration is for advanced use cases and is usually not necessary. If you choose to configure the properties below, you will override any changes to the Next.js defaults in future updates.

Device Sizes

If you know the expected device widths of your users, you can specify a list of device width breakpoints using the `deviceSizes` property in `next.config.js`. These widths are used when the `next/legacy/image` component uses `layout="responsive"` or `layout="fill"` to ensure the correct image is served for user's device.

If no configuration is provided, the default below is used.

JS next.config.js

```
1 module.exports = {
2   images: {
3     deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
4   },
5 };
```

Image Sizes

You can specify a list of image widths using the `images.imageSizes` property in your `next.config.js` file. These widths are concatenated with the array of `device sizes` to form the full array of sizes used to generate image [srcset](#) ↗ s.

The reason there are two separate lists is that `imageSizes` is only used for images which provide a `sizes` prop, which indicates that the image is less than the full width of the screen. **Therefore, the sizes in `imageSizes` should all be smaller than the smallest size in `deviceSizes`.**

If no configuration is provided, the default below is used.

JS next.config.js

```
1 module.exports = {
2   images: {
3     imageSizes: [16, 32, 48, 64, 96, 128, 256, 384],
4   },
5 };
```

Acceptable Formats

The default [Image Optimization API](#) will automatically detect the browser's supported image formats via the request's `Accept` header.

If the `Accept` head matches more than one of the configured formats, the first match in the array is used. Therefore, the array order matters. If there is no match (or the source image is [animated](#)), the Image Optimization API will fallback to the original image's format.

If no configuration is provided, the default below is used.

JS next.config.js

```
1 module.exports = {
2   images: {
3     formats: ['image/webp'],
4   },
5 };
```

You can enable AVIF support with the following configuration.

JS next.config.js

```
1 module.exports = {
2   images: {
3     formats: ['image/avif', 'image/webp'],
4   },
5 };
```

Note: AVIF generally takes 20% longer to encode but it compresses 20% smaller compared to WebP. This means that the first time an image is requested, it will typically be slower and then subsequent requests that are cached will be faster.

Caching Behavior

The following describes the caching algorithm for the default [loader](#). For all other loaders, please refer to your cloud provider's documentation.

Images are optimized dynamically upon request and stored in the `<distDir>/cache/images` directory. The optimized image file will be served for subsequent requests until the expiration is reached. When a request is made that matches a cached but expired file, the expired image is served stale immediately. Then the image is optimized again in the background (also called revalidation) and saved to the cache with the new expiration date.

The cache status of an image can be determined by reading the value of the `x-nextjs-cache` (`x-vercel-cache` when deployed on Vercel) 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

The expiration (or rather Max Age) is defined by either the [minimumCacheTTL](#) configuration or the upstream image `Cache-Control` header, whichever is larger. Specifically, the `max-age` value of the `Cache-Control` header is used. If both `s-maxage` and `max-age` are found, then `s-maxage` is preferred. The `max-age` is also passed-through to any downstream clients including CDNs and browsers.

- You can configure [minimumCacheTTL](#) to increase the cache duration when the upstream image does not include `Cache-Control` header or the value is very low.
- You can configure [deviceSizes](#) and [imageSizes](#) to reduce the total number of possible generated images.
- You can configure [formats](#) to disable multiple formats in favor of a single image format.

Minimum Cache TTL

You can configure the Time to Live (TTL) in seconds for cached optimized images. In many cases, it's better to use a [Static Image Import](#) which will automatically hash the file contents and cache the image forever with a `Cache-Control` header of `immutable`.

 next.config.js

```
1 module.exports = {
2   images: {
```



```
3     minimumCacheTTL: 60,
4   },
5 };
```

The expiration (or rather Max Age) of the optimized image is defined by either the `minimumCacheTTL` or the upstream image `Cache-Control` header, whichever is larger.

If you need to change the caching behavior per image, you can configure `headers` to set the `Cache-Control` header on the upstream image (e.g. `/some-asset.jpg`, not `/_next/image` itself).

There is no mechanism to invalidate the cache at this time, so its best to keep `minimumCacheTTL` low. Otherwise you may need to manually change the `src` prop or delete `<distDir>/cache/images`.

Disable Static Imports

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property.

In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

You can disable static image imports inside your `next.config.js`:

JS next.config.js

```
1 module.exports = {
2   images: {
3     disableStaticImages: true,
4   },
5 };
```

Dangerously Allow SVG

The default `loader` does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper [Content Security Policy \(CSP\) headers](#).

If you need to serve SVG images with the default Image Optimization API, you can set `dangerouslyAllowSVG` inside your `next.config.js`:

JS next.config.js

```
1 module.exports = {
```

```
2   images: {
3     dangerouslyAllowSVG: true,
4     contentDispositionType: 'attachment',
5     contentSecurityPolicy: "default-src 'self'; script-src 'none'; sandbox;",
6   },
7 }
```

In addition, it is strongly recommended to also set `contentDispositionType` to force the browser to download the image, as well as `contentSecurityPolicy` to prevent scripts embedded in the image from executing.

Animated Images

The default [loader](#) will automatically bypass Image Optimization for animated images and serve the image as-is.

Auto-detection for animated files is best-effort and supports GIF, APNG, and WebP. If you want to explicitly bypass Image Optimization for a given animated image, use the [unoptimized](#) prop.

Version History

Version	Changes
v13.0.0	<code>next/image</code> renamed to <code>next/legacy/image</code>

> Menu

Pages Router > ... > Components > <Link>

<Link>

► Examples

<Link> is a React component that extends the HTML <a> element to provide **prefetching** and client-side navigation between routes. It is the primary way to navigate between routes in Next.js.

For an example, consider a `pages` directory with the following files:

- `pages/index.js`
- `pages/about.js`
- `pages/blog/[slug].js`

We can have a link to each of these pages like so:

```
1 import Link from 'next/link';
2
3 function Home() {
4   return (
5     <ul>
6       <li>
7         <Link href="/">Home</Link>
8       </li>
9       <li>
10        <Link href="/about">About Us</Link>
11      </li>
12      <li>
13        <Link href="/blog/hello-world">Blog Post</Link>
14      </li>
15    </ul>
16  );
17}
18
19 export default Home;
```

Props

Here's a summary of the props available for the Link Component:

Prop	Example	Type	Required
href	href="/dashboard"	String or Object	Yes
replace	replace={false}	Boolean	-
prefetch	prefetch={false}	Boolean	-

Good to know: `<a>` tag attributes such as `className` or `target="_blank"` can be added to `<Link>` as props and will be passed to the underlying `<a>` element.

href (required)

The path or URL to navigate to.

```
<Link href="/dashboard">Dashboard</Link>
```

`href` can also accept an object, for example:

```
1 // Navigate to /about?name=test
2 <Link
3   href={{
4     pathname: '/about',
5     query: { name: 'test' },
6   }}
7 >
8   About
9 </Link>
```

replace

Defaults to false. When `true`, `next/link` will replace the current history state instead of adding a new URL into the [browser's history](#) stack.

app/page.tsx

```
1 import Link from 'next/link';
```

```
2
3 export default function Page() {
4   return (
5     <Link href="/dashboard" replace>
6       Dashboard
7     </Link>
8   );
9 }
```

prefetch

Defaults to `true`. When `true`, `next/link` will prefetch the page (denoted by the `href`) in the background. This is useful for improving the performance of client-side navigations. Any `<Link />` in the viewport (initially or through scroll) will be preloaded.

Prefetch can be disabled by passing `prefetch={false}`. Prefetching is only enabled in production.

app/page.tsx

```
1 import Link from 'next/link';
2
3 export default function Page() {
4   return (
5     <Link href="/dashboard" prefetch={false}>
6       Dashboard
7     </Link>
8   );
9 }
```

Other Props

legacyBehavior

An `<a>` element is no longer required as a child of `<Link>`. Add the `legacyBehavior` prop to use the legacy behavior or remove the `<a>` to upgrade. A [codemod is available](#) to automatically upgrade your code.

Note: when `legacyBehavior` is not set to `true`, all `anchor` ↗ tag properties can be passed to `next/link` as well such as, `className`, `onClick`, etc.

passHref

Forces `Link` to send the `href` property to its child. Defaults to `false`

scroll

Scroll to the top of the page after a navigation. Defaults to `true`

shallow

Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to `false`

locale

The active locale is automatically prepended. `locale` allows for providing a different locale. When `false` `href` has to include the locale as the default behavior is disabled.

Examples

Linking to Dynamic Routes

For dynamic routes, it can be handy to use template literals to create the link's path.

For example, you can generate a list of links to the dynamic route `pages/blog/[slug].js`

JS `pages/blog/index.js`

```
1 import Link from 'next/link';
2
3 function Posts({ posts }) {
4   return (
5     <ul>
6       {posts.map((post) => (
7         <li key={post.id}>
8           <Link href={`/blog/${post.slug}`}>{post.title}</Link>
9         </li>
10      )));
11    </ul>
12  );
13}
14
15 export default Posts;
```

If the child is a custom component that wraps an `<a>` tag

If the child of `Link` is a custom component that wraps an `<a>` tag, you must add `passHref` to `Link`. This is necessary if you're using libraries like [styled-components](#)[↗]. Without this, the `<a>` tag will not have the `href` attribute, which hurts your site's accessibility and might affect SEO. If you're using [ESLint](#), there is a built-in rule `next/link-passhref` to ensure correct usage of `passHref`.

```
1 import Link from 'next/link';
2 import styled from 'styled-components';
3
4 // This creates a custom component that wraps an <a> tag
5 const RedLink = styled.a`  

6   color: red;  

7 `;
8
9 function NavLink({ href, name }) {
10   return (
11     <Link href={href} passHref legacyBehavior>
12       <RedLink>{name}</RedLink>
13     </Link>
14   );
15 }
16
17 export default NavLink;
```

- If you're using [emotion](#)[↗]'s JSX pragma feature (`@jsx jsx`), you must use `passHref` even if you use an `<a>` tag directly.
- The component should support `onClick` property to trigger navigation correctly

If the child is a functional component

If the child of `Link` is a functional component, in addition to using `passHref` and `legacyBehavior`, you must wrap the component in [React.forwardRef](#)[↗]:

```
1 import Link from 'next/link';
2
3 // `onClick`, `href`, and `ref` need to be passed to the DOM element
4 // for proper handling
5 const MyButton = React.forwardRef(({ onClick, href }, ref) => {
6   return (
7     <a href={href} onClick={onClick} ref={ref}>
8       Click Me
9     </a>
10    );
11  });
12
13 function Home() {
14   return (
15     <Link href="/about" passHref legacyBehavior>
16       <MyButton />
```

```
17     </Link>
18   );
19 }
20
21 export default Home;
```

With URL Object

`Link` can also receive a URL object and it will automatically format it to create the URL string. Here's how to do it:

```
1 import Link from 'next/link';
2
3 function Home() {
4   return (
5     <ul>
6       <li>
7         <Link
8           href={{
9             pathname: '/about',
10            query: { name: 'test' },
11          }}
12        >
13          About us
14        </Link>
15      </li>
16      <li>
17        <Link
18          href={{
19            pathname: '/blog/[slug]',
20            query: { slug: 'my-post' },
21          }}
22        >
23          Blog Post
24        </Link>
25      </li>
26    </ul>
27  );
28}
29
30 export default Home;
```

The above example has a link to:

- A predefined route: `/about?name=test`
- A dynamic route: `/blog/my-post`

You can use every property as defined in the [Node.js URL module documentation ↗](#).

Replace the URL instead of push

The default behavior of the `Link` component is to `push` a new URL into the `history` stack. You can use the `replace` prop to prevent adding a new entry, as in the following example:

```
1 <Link href="/about" replace>
2   About us
3 </Link>
```

Disable scrolling to the top of the page

The default behavior of `Link` is to scroll to the top of the page. When there is a hash defined it will scroll to the specific id, like a normal `<a>` tag. To prevent scrolling to the top / hash `scroll={false}` can be added to `Link`:

```
1 <Link href="#hashid" scroll={false}>
2   Disables scrolling to the top
3 </Link>
```

Middleware

It's common to use [Middleware](#) for authentication or other purposes that involve rewriting the user to a different page. In order for the `<Link />` component to properly prefetch links with rewrites via Middleware, you need to tell Next.js both the URL to display and the URL to prefetch. This is required to avoid un-necessary fetches to middleware to know the correct route to prefetch.

For example, if you have want to serve a `/dashboard` route that has authenticated and visitor views, you may add something similar to the following in your Middleware to redirect the user to the correct page:

js middleware.js



```
1 export function middleware(req) {
2   const nextUrl = req.nextUrl;
3   if (nextUrl.pathname === '/dashboard') {
4     if (req.cookies.authToken) {
5       return NextResponse.rewrite(new URL('/auth/dashboard', req.url));
6     } else {
7       return NextResponse.rewrite(new URL('/public/dashboard', req.url));
8     }
9   }
10 }
```

In this case, you would want to use the following code in your `<Link />` component:

```
1 import Link from 'next/link';
2 import useIsAuthed from './hooks/useIsAuthed';
3
4 export default function Page() {
5   const isAuthenticated = useIsAuthed();
6   const path = isAuthenticated ? '/auth/dashboard' : '/dashboard';
7   return (
8     <Link as="/dashboard" href={path}>
9       Dashboard
10      </Link>
11    );
12 }
```

Note: If you're using [Dynamic Routes](#), you'll need to adapt your `as` and `href` props. For example, if you have a Dynamic Route like `/dashboard/[user]` that you want to present differently via middleware, you would write:

```
<Link href={{ pathname: '/dashboard/authed/[user]', query: { user: username } }}>
  as="/dashboard/[user]">Profile</Link>
```

.

> Menu

Pages Router > ... > Components > <Script>

<Script>

This API reference will help you understand how to use [props](#) available for the Script Component. For features and usage, please see the [Optimizing Scripts](#) page.

TS app/dashboard/page.tsx

```
1 import Script from 'next/script';
2
3 export default function Dashboard() {
4   return (
5     <>
6       <Script src="https://example.com/script.js" />
7     </>
8   );
9 }
```

Props

Here's a summary of the props available for the Script Component:

Prop	Example	Type	Required
src	src="http://example.com/script"	String	Required unless inline script is used
strategy	strategy="lazyOnload"	String	-
onLoad	onLoad={onLoadFunc}	Function	-
onReady	onReady={onReadyFunc}	Function	-
onError	onError={onErrorFunc}	Function	-

Required Props

The `<Script />` component requires the following properties.

src

A path string specifying the URL of an external script. This can be either an absolute external URL or an internal path. The `src` property is required unless an inline script is used.

Optional Props

The `<Script />` component accepts a number of additional properties beyond those which are required.

strategy

The loading strategy of the script. There are four different strategies that can be used:

- `beforeInteractive` : Load before any Next.js code and before any page hydration occurs.
- `afterInteractive` : (**default**) Load early but after some hydration on the page occurs.
- `lazyOnload` : Load during browser idle time.
- `worker` : (experimental) Load in a web worker.

beforeInteractive

Scripts that load with the `beforeInteractive` strategy are injected into the initial HTML from the server, downloaded before any Next.js module, and executed in the order they are placed before *any* hydration occurs on the page.

Scripts denoted with this strategy are preloaded and fetched before any first-party code, but their execution does not block page hydration from occurring.

`beforeInteractive` scripts must be placed inside the root layout (`app/layout.tsx`) and are designed to load scripts that are needed by the entire site (i.e. the script will load when any page in the application has been loaded server-side).

This strategy should only be used for critical scripts that need to be fetched before any part of the page becomes interactive.

```
1 import { Html, Head, Main, NextScript } from 'next/document';
2 import Script from 'next/script';
3
4 export default function Document() {
5   return (
6     <Html>
7       <Head />
8       <body>
9         <Main />
10        <NextScript />
11        <Script
12          src="https://example.com/script.js"
13          strategy="beforeInteractive"
14        />
15      </body>
16    </Html>
17  );
18}
```

Good to know: Scripts with `beforeInteractive` will always be injected inside the `head` of the HTML document regardless of where it's placed in the component.

Some examples of scripts that should be loaded as soon as possible with `beforeInteractive` include:

- Bot detectors
- Cookie consent managers

afterInteractive

Scripts that use the `afterInteractive` strategy are injected into the HTML client-side and will load after some (or all) hydration occurs on the page. **This is the default strategy** of the Script component and should be used for any script that needs to load as soon as possible but not before any first-party Next.js code.

`afterInteractive` scripts can be placed inside of any page or layout and will only load and execute when that page (or group of pages) is opened in the browser.

js app/page.js

```
1 import Script from 'next/script';
2
3 export default function Page() {
4   return (
5     <>
6       <Script src="https://example.com/script.js" strategy="afterInteractive" />
7     </>
8   );
}
```

Some examples of scripts that are good candidates for `afterInteractive` include:

- Tag managers
- Analytics

lazyOnload

Scripts that use the `lazyOnload` strategy are injected into the HTML client-side during browser idle time and will load after all resources on the page have been fetched. This strategy should be used for any background or low priority scripts that do not need to load early.

`lazyOnload` scripts can be placed inside of any page or layout and will only load and execute when that page (or group of pages) is opened in the browser.

js app/page.js

```
1 import Script from 'next/script';
2
3 export default function Page() {
4   return (
5     <>
6       <Script src="https://example.com/script.js" strategy="lazyOnload" />
7     </>
8   );
9 }
```

Examples of scripts that do not need to load immediately and can be fetched with `lazyOnload` include:

- Chat support plugins
- Social media widgets

worker

Warning: The `worker` strategy is not yet stable and does not yet work with the `app` directory. Use with caution.

Scripts that use the `worker` strategy are off-loaded to a web worker in order to free up the main thread and ensure that only critical, first-party resources are processed on it. While this strategy can be used for any script, it is an advanced use case that is not guaranteed to support all third-party scripts.

To use `worker` as a strategy, the `nextScriptWorkers` flag must be enabled in `next.config.js`:



```

1 module.exports = {
2   experimental: {
3     nextScriptWorkers: true,
4   },
5 };

```

`worker` scripts can **only currently be used in the `pages/` directory**:



```

1 import Script from 'next/script';
2
3 export default function Home() {
4   return (
5     <>
6       <Script src="https://example.com/script.js" strategy="worker" />
7     </>
8   );
9 }

```

onLoad

Warning: `onLoad` does not yet work with Server Components and can only be used in Client Components. Further, `onLoad` can't be used with `beforeInteractive` – consider using `onReady` instead.

Some third-party scripts require users to run JavaScript code once after the script has finished loading in order to instantiate content or call a function. If you are loading a script with either `afterInteractive` or `lazyOnload` as a loading strategy, you can execute code after it has loaded using the `onLoad` property.

Here's an example of executing a lodash method only after the library has been loaded.



```

1 'use client';
2
3 import Script from 'next/script';
4
5 export default function Page() {
6   return (
7     <>
8       <Script
9         src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.20/lodash.min.js"
10        onLoad={() => {

```

```
11         console.log(_.sample([1, 2, 3, 4]));
12     })
13   />
14 </>
15 );
16 }
```

onReady

Warning: `onReady` does not yet work with Server Components and can only be used in Client Components.

Some third-party scripts require users to run JavaScript code after the script has finished loading and every time the component is mounted (after a route navigation for example). You can execute code after the script's load event when it first loads and then after every subsequent component re-mount using the `onReady` property.

Here's an example of how to re-instantiate a Google Maps JS embed every time the component is mounted:

```
1 import { useRef } from 'react';
2 import Script from 'next/script';
3
4 export default function Page() {
5   const mapRef = useRef();
6
7   return (
8     <PagesOnly>
9       <div ref={mapRef}></div>
10      <Script
11        id="google-maps"
12        src="https://maps.googleapis.com/maps/api/js"
13        onReady={() => {
14          new google.maps.Map(mapRef.current, {
15            center: { lat: -34.397, lng: 150.644 },
16            zoom: 8,
17          });
18        }}
19      />
20    </>
21  );
22}
```

onError

Warning: `onError` does not yet work with Server Components and can only be used in Client Components. `onError` cannot be used with the `beforeInteractive` loading strategy.

Sometimes it is helpful to catch when a script fails to load. These errors can be handled with the `onError` property:

```
1 import Script from 'next/script';
2
3 export default function Page() {
4   return (
5     <>
6       <Script
7         src="https://example.com/script.js"
8         onError={(e: Error) => {
9           console.error('Script failed to load', e);
10        }}
11      />
12      </>
13    );
14 }
```

Version History

Version	Changes
v13.0.0	<code>beforeInteractive</code> and <code>afterInteractive</code> is modified to support <code>app</code>
v12.2.4	<code>onReady</code> prop added.
v12.2.2	Allow <code>next/script</code> with <code>beforeInteractive</code> to be placed in <code>_document</code> .
v11.0.0	<code>next/script</code> introduced.

> Menu

Pages Router > API Reference > create-next-app

create-next-app

The easiest way to get started with Next.js is by using `create-next-app`. This CLI tool enables you to quickly start building a new Next.js application, with everything set up for you. You can create a new app using the default Next.js template, or by using one of the [official Next.js examples ↗](#). To get started, use the following command:

Interactive

You can create a new project interactively by running:

> Terminal

```
1 npx create-next-app@latest
2
3 yarn create next-app
4
5 pnpm create next-app
```

You will then be asked the following prompts:

> Terminal

```
1 What is your project named? my-app
2 Would you like to add TypeScript with this project? Y/N
3 Would you like to use ESLint with this project? Y/N
4 Would you like to use Tailwind CSS with this project? Y/N
5 Would you like to use the `src/ directory` with this project? Y/N
6 What import alias would you like configured? `@/*`
```

Once you've answered the prompts, a new project will be created with the correct configuration depending on your answers.

Non-interactive

You can also pass command line arguments to set up a new project non-interactively.

Further, you can negate default options by prefixing them with `--no-` (e.g. `--no-eslint`).

See `create-next-app --help`:

>_ Terminal



```
1 Usage: create-next-app <project-directory> [options]
2
3 Options:
4   -V, --version                                output the version number
5   --ts, --typescript
6
7     Initialize as a TypeScript project. (default)
8
9   --js, --javascript
10
11    Initialize as a JavaScript project.
12
13   --tailwind
14
15    Initialize with Tailwind CSS config. (default)
16
17   --eslint
18
19    Initialize with ESLint config.
20
21   --src-dir
22
23    Initialize inside a `src/` directory.
24
25   --import-alias <alias-to-configure>
26
27    Specify import alias to use (default "@/*").
28
29   --use-npm
30
31    Explicitly tell the CLI to bootstrap the app using npm
32
33   --use-pnpm
34
35    Explicitly tell the CLI to bootstrap the app using pnpm
36
37   -e, --example [name]|[github-url]
38
39    An example to bootstrap the app with. You can use an example name
40    from the official Next.js repo or a GitHub URL. The URL can use
41    any branch and/or subdirectory
42
43   --example-path <path-to-example>
44
45    In a rare case, your GitHub URL might contain a branch name with
```

```
46      a slash (e.g. bug/fix-1) and the path to the example (e.g. foo/bar).  
47      In this case, you must specify the path to the example separately:  
48      --example-path foo/bar  
49  
50      --reset-preferences  
51  
52      Explicitly tell the CLI to reset any stored preferences  
53  
54      -h, --help                                output usage information
```

Why use Create Next App?

`create-next-app` allows you to create a new Next.js app within seconds. It is officially maintained by the creators of Next.js, and includes a number of benefits:

- **Interactive Experience:** Running `npx create-next-app@latest` (with no arguments) launches an interactive experience that guides you through setting up a project.
- **Zero Dependencies:** Initializing a project is as quick as one second. Create Next App has zero dependencies.
- **Offline Support:** Create Next App will automatically detect if you're offline and bootstrap your project using your local package cache.
- **Support for Examples:** Create Next App can bootstrap your application using an example from the Next.js examples collection (e.g. `npx create-next-app --example api-routes`).
- **Tested:** The package is part of the Next.js monorepo and tested using the same integration test suite as Next.js itself, ensuring it works as expected with every release.

> Menu

Pages Router > API Reference > Edge Runtime

Edge Runtime

The Next.js Edge Runtime is based on standard Web APIs, it supports the following APIs:

Network APIs

API	Description
fetch ↗	Fetches a resource
Request ↗	Represents an HTTP request
Response ↗	Represents an HTTP response
Headers ↗	Represents HTTP headers
FetchEvent ↗	Represents a fetch event
addEventListener ↗	Adds an event listener
FormData ↗	Represents form data
File ↗	Represents a file
Blob ↗	Represents a blob
URLSearchParams ↗	Represents URL search parameters

Encoding APIs

API	Description
TextEncoder ↗	Encodes a string into a Uint8Array

API	Description
TextDecoder ↗	Decodes a Uint8Array into a string
atob ↗	Decodes a base-64 encoded string
btoa ↗	Encodes a string in base-64

Stream APIs

API	Description
ReadableStream ↗	Represents a readable stream
WritableStream ↗	Represents a writable stream
WritableStreamDefaultWriter ↗	Represents a writer of a WritableStream
TransformStream ↗	Represents a transform stream
ReadableStreamDefaultReader ↗	Represents a reader of a ReadableStream
ReadableStreamBYOBReader ↗	Represents a reader of a ReadableStream

Crypto APIs

API	Description
crypto ↗	Provides access to the cryptographic functionality of the platform
SubtleCrypto ↗	Provides access to common cryptographic primitives, like hashing, signing, encryption or decryption
CryptoKey ↗	Represents a cryptographic key

Web Standard APIs

API	Description
AbortController ↗	Allows you to abort one or more DOM requests as and when desired

API	Description
DOMException ↗	Represents an error that occurs in the DOM
structuredClone ↗	Creates a deep copy of a value
URLPattern ↗	Represents a URL pattern
Array ↗	Represents an array of values
ArrayBuffer ↗	Represents a generic, fixed-length raw binary data buffer
Atomics ↗	Provides atomic operations as static methods
BigInt ↗	Represents a whole number with arbitrary precision
BigInt64Array ↗	Represents a typed array of 64-bit signed integers
BigUint64Array ↗	Represents a typed array of 64-bit unsigned integers
Boolean ↗	Represents a logical entity and can have two values: <code>true</code> and <code>false</code>
clearInterval ↗	Cancels a timed, repeating action which was previously established by a call to <code>setInterval()</code>
clearTimeout ↗	Cancels a timed, repeating action which was previously established by a call to <code>setTimeout()</code>
console ↗	Provides access to the browser's debugging console
DataView ↗	Represents a generic view of an <code>ArrayBuffer</code>
Date ↗	Represents a single moment in time in a platform-independent format
decodeURI ↗	Decodes a Uniform Resource Identifier (URI) previously created by <code>encodeURI</code> or by a similar routine
decodeURIComponent ↗	Decodes a Uniform Resource Identifier (URI) component previously created by <code>encodeURIComponent</code> or by a similar routine
encodeURI ↗	Encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character
encodeURIComponent ↗	Encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character
Error ↗	Represents an error when trying to execute a statement or accessing a property
EvalError ↗	Represents an error that occurs regarding the global function <code>eval()</code>
Float32Array ↗	Represents a typed array of 32-bit floating point numbers
Float64Array ↗	Represents a typed array of 64-bit floating point numbers

API	Description
Function ↗	Represents a function
Infinity ↗	Represents the mathematical Infinity value
Int8Array ↗	Represents a typed array of 8-bit signed integers
Int16Array ↗	Represents a typed array of 16-bit signed integers
Int32Array ↗	Represents a typed array of 32-bit signed integers
Intl ↗	Provides access to internationalization and localization functionality
isFinite ↗	Determines whether a value is a finite number
isNaN ↗	Determines whether a value is <code>Nan</code> or not
JSON ↗	Provides functionality to convert JavaScript values to and from the JSON format
Map ↗	Represents a collection of values, where each value may occur only once
Math ↗	Provides access to mathematical functions and constants
Number ↗	Represents a numeric value
Object ↗	Represents the object that is the base of all JavaScript objects
parseFloat ↗	Parses a string argument and returns a floating point number
parseInt ↗	Parses a string argument and returns an integer of the specified radix
Promise ↗	Represents the eventual completion (or failure) of an asynchronous operation, and its resulting value
Proxy ↗	Represents an object that is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc)
RangeError ↗	Represents an error when a value is not in the set or range of allowed values
ReferenceError ↗	Represents an error when a non-existent variable is referenced
Reflect ↗	Provides methods for interceptable JavaScript operations
RegExp ↗	Represents a regular expression, allowing you to match combinations of characters
Set ↗	Represents a collection of values, where each value may occur only once
setInterval ↗	Repeatedly calls a function, with a fixed time delay between each call
setTimeout ↗	Calls a function or evaluates an expression after a specified number of milliseconds

API	Description
SharedArrayBuffer ↗	Represents a generic, fixed-length raw binary data buffer
String ↗	Represents a sequence of characters
Symbol ↗	Represents a unique and immutable data type that is used as the key of an object property
SyntaxError ↗	Represents an error when trying to interpret syntactically invalid code
TypeError ↗	Represents an error when a value is not of the expected type
Uint8Array ↗	Represents a typed array of 8-bit unsigned integers
Uint8ClampedArray ↗	Represents a typed array of 8-bit unsigned integers clamped to 0-255
Uint32Array ↗	Represents a typed array of 32-bit unsigned integers
URIError ↗	Represents an error when a global URI handling function was used in a wrong way
URL ↗	Represents an object providing static methods used for creating object URLs
URLSearchParams ↗	Represents a collection of key/value pairs
WeakMap ↗	Represents a collection of key/value pairs in which the keys are weakly referenced
WeakSet ↗	Represents a collection of objects in which each object may occur only once
WebAssembly ↗	Provides access to WebAssembly

Next.js Specific Polyfills

- [AsyncLocalStorage](#) ↗

Environment Variables

You can use `process.env` to access [Environment Variables](#) for both `next dev` and `next build`.

Unsupported APIs

The Edge Runtime has some restrictions including:

- Native Node.js APIs **are not supported**. For example, you can't read or write to the filesystem.
- `node_modules` *can* be used, as long as they implement ES Modules and do not use native Node.js APIs.
- Calling `require` directly is **not allowed**. Use ES Modules instead.

The following JavaScript language features are disabled, and **will not work**:

API	Description
<code>eval</code> ↗	Evaluates JavaScript code represented as a string
<code>new Function(evalString)</code> ↗	Creates a new function with the code provided as an argument
<code>WebAssembly.compile</code> ↗	Compiles a WebAssembly module from a buffer source
<code>WebAssembly.instantiate</code> ↗	Compiles and instantiates a WebAssembly module from a buffer source

In rare cases, your code could contain (or import) some dynamic code evaluation statements which *can not be reached at runtime* and which can not be removed by treeshaking. You can relax the check to allow specific files with your Middleware or Edge API Route exported configuration:

```
1  export const config = {
2    runtime: 'edge', // for Edge API Routes only
3    unstable_allowDynamic: [
4      // allows a single file
5      '/lib/utilities.js',
6      // use a glob to allow anything in the function-bind 3rd party module
7      '/node_modules/function-bind/**',
8    ],
9  };
```

`unstable_allowDynamic` is a [glob ↗](#), or an array of globs, ignoring dynamic code evaluation for specific files. The globs are relative to your application root folder.

Be warned that if these statements are executed on the Edge, *they will throw and cause a runtime error*.

> Menu

Pages Router > API Reference > Functions

Functions

[getInitialProps](#)

Fetch dynamic data on the server for your React component with `getInitialProps`.

[getServerSideProps](#)

API reference for ``getServerSideProps``. Learn how to fetch data on each request with Next.js.

[getStaticPaths](#)

API reference for ``getStaticPaths``. Learn how to fetch data and generate static pages with ``getStaticPaths``.

[getStaticProps](#)

API reference for ``getStaticProps``. Learn how to use ``getStaticProps`` to generate static pages with Next.js.

[NextRequest and NextResponse](#)

Learn about the server-only helpers for Middleware and Edge API Routes.

[useAmp](#)

Enable AMP in a page, and control the way Next.js adds AMP to the page with the AMP config.

[useReportWebVitals](#)

useReportWebVitals

[useRouter](#)

Learn more about the API of the Next.js Router, and access the router instance in your page with the useRouter hook.

> Menu

Pages Router > ... > Functions > getInitialProps

getInitialProps

Note: `getInitialProps` is a legacy API. We recommend using `getStaticProps` or `getServerSideProps` instead.

`getInitialProps` is an `async` function that can be added to the default exported React component for the page. It will run on both the server-side and again on the client-side during page transitions. The result of the function will be forwarded to the React component as `props`.

TS pages/index.tsx

```
1 import { NextPageContext } from 'next';
2
3 Page.getInitialProps = async (ctx: NextPageContext) => {
4   const res = await fetch('https://api.github.com/repos/vercel/next.js');
5   const json = await res.json();
6   return { stars: json.stargazers_count };
7 }
8
9 export default function Page({ stars }: { stars: number }) {
10   return stars;
11 }
```

Good to know:

- Data returned from `getInitialProps` is serialized when server rendering. Ensure the returned object from `getInitialProps` is a plain `Object`, and not using `Date`, `Map` or `Set`.
- For the initial page load, `getInitialProps` will run on the server only. `getInitialProps` will then also run on the client when navigating to a different route with the `next/link` component or by using `next/router`.
- If `getInitialProps` is used in a custom `_app.js`, and the page being navigated to is using `getServerSideProps`, then `getInitialProps` will also run on the server.

Context Object

`getInitialProps` receives a single argument called `context`, which is an object with the following properties:

Name	Description
<code>pathname</code>	Current route, the path of the page in <code>/pages</code>
<code>query</code>	Query string of the URL, parsed as an object
<code>asPath</code>	<code>String</code> of the actual path (including the query) shown in the browser
<code>req</code>	HTTP request object ↗ (server only)
<code>res</code>	HTTP response object ↗ (server only)
<code>err</code>	Error object if any error is encountered during the rendering

Caveats

- `getInitialProps` can only be used in `pages/` top level files, and not in nested components. To have nested data fetching at the component level, consider exploring the [App Router](#).
- Regardless of whether your route is static or dynamic, any data returned from `getInitialProps` as `props` will be able to be examined on the client-side in the initial HTML. This is to allow the page to be [hydrated ↗](#) correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

> Menu

Pages Router > ... > Functions > getServerSideProps

getServerSideProps

When exporting a function called `getServerSideProps` (Server-Side Rendering) from a page, Next.js will pre-render this page on each request using the data returned by `getServerSideProps`. This is useful if you want to fetch data that changes often, and have the page update to show the most current data.

TS pages/index.tsx

```
1 import type { InferGetServerSidePropsType, GetServerSideProps } from 'next';
2
3 type Repo = {
4   name: string;
5   stargazers_count: number;
6 };
7
8 export const getServerSideProps: GetServerSideProps<{
9   repo: Repo;
10 }> = async () => {
11   const res = await fetch('https://api.github.com/repos/vercel/next.js');
12   const repo = await res.json();
13   return { props: { repo } };
14 };
15
16 export default function Page({
17   repo,
18 }: InferGetServerSidePropsType<typeof getServerSideProps>) {
19   return repo.stargazers_count;
20 }
```

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

Context parameter

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

Name	Description
<code>params</code>	If this page uses a dynamic route , <code>params</code> contains the route parameters. If the page name is <code>[id].js</code> , then <code>params</code> will look like <code>{ id: ... }</code> .
<code>req</code>	The HTTP IncomingMessage object , with an additional <code>cookies</code> prop, which is an object with string keys mapping to string values of cookies.
<code>res</code>	The HTTP response object .
<code>query</code>	An object representing the query string, including dynamic route parameters.
<code>preview</code>	(Deprecated for <code>draftMode</code>) <code>preview</code> is <code>true</code> if the page is in the Preview Mode and <code>false</code> otherwise.
<code>previewData</code>	(Deprecated for <code>draftMode</code>) The <code>preview</code> data set by <code>setPreviewData</code> .
<code>draftMode</code>	<code>draftMode</code> is <code>true</code> if the page is in the Draft Mode and <code>false</code> otherwise.
<code>resolvedUrl</code>	A normalized version of the request <code>URL</code> that strips the <code>_next/data</code> prefix for client transitions and includes original query values.
<code>locale</code>	Contains the active locale (if enabled).
<code>locales</code>	Contains all supported locales (if enabled).
<code>defaultLocale</code>	Contains the configured default locale (if enabled).

getServerSideProps return values

The `getServerSideProps` function should return an object with **any one of the following** properties:

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`.

```
1 export async function getServerSideProps(context) {  
2   return {  
3     props: { message: `Next.js is awesome` }, // will be passed to the page component as  
4   };  
5 }
```

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.

```
1 export async function getServerSideProps(context) {
2   const res = await fetch(`https://.../data`);
3   const data = await res.json();
4
5   if (!data) {
6     return {
7       notFound: true,
8     };
9   }
10
11   return {
12     props: { data }, // will be passed to the page component as props
13   };
14 }
```

redirect

The `redirect` object allows 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.

```
1 export async function getServerSideProps(context) {
2   const res = await fetch(`https://.../data`);
3   const data = await res.json();
4
5   if (!data) {
6     return {
7       redirect: {
8         destination: '/',
9         permanent: false,
10        },
11      };
12    }
13
14   return {
15     props: {}, // will be passed to the page component as props
16   };
17 }
```

Version History

Version	Changes
v13.4.0	App Router is now stable with simplified data fetching
v10.0.0	<code>locale</code> , <code>locales</code> , <code>defaultLocale</code> , and <code>notFound</code> options added.
v9.3.0	<code>getServerSideProps</code> introduced.

> Menu

Pages Router > ... > Functions > getStaticPaths

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`.

TS pages/repo/[name].tsx

```
1 import type {
2   InferGetStaticPropsType,
3   GetStaticProps,
4   GetStaticPaths,
5 } from 'next';
6
7 type Repo = {
8   name: string;
9   stargazers_count: number;
10};
11
12 export const getStaticPaths: GetStaticPaths = async () => {
13   return {
14     paths: [
15       {
16         params: {
17           name: 'next.js',
18         },
19       }, // See the "paths" section below
20     ],
21     fallback: true, // false or "blocking"
22   };
23 };
24
25 export const getStaticProps: GetStaticProps<{
26   repo: Repo;
27 }> = async () => {
28   const res = await fetch('https://api.github.com/repos/vercel/next.js');
29   const repo = await res.json();
30   return { props: { repo } };
31 };
32
33 export default function Page({
34   repo,
35 }: InferGetStaticPropsType<typeof getStaticProps>) {
```

```
36     return repo.stargazers_count;
37 }
```

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`:

```
1  return {
2    paths: [
3      { params: { id: '1' } },
4      {
5        params: { id: '2' },
6        // with i18n configured the locale for the path can be returned as well
7        locale: "en",
8      },
9    ],
10   fallback: ...
11 }
```

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](#).

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



```
1 function Post({ post }) {
2   // Render post...
3 }
4
5 // This function gets called at build time
6 export async function getStaticPaths() {
7   // Call an external API endpoint to get posts
8   const res = await fetch('https://.../posts');
9   const posts = await res.json();
10
11  // Get the paths we want to pre-render based on posts
12  const paths = posts.map((post) => ({
13    params: { id: post.id },
14  }));
15
16  // We'll pre-render only these paths at build time.
17  // { fallback: false } means other routes should 404.
18  return { paths, fallback: false };
19 }
20
21 // This also gets called at build time
22 export async function getStaticProps({ params }) {
23   // params contains the post `id`.
24   // If the route is like /posts/1, then params.id is 1
25   const res = await fetch(`https://.../posts/${params.id}`);
26   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`:

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

```
1 import { useRouter } from 'next/router';
2
3 function Post({ post }) {
```

```

4  const router = useRouter();
5
6  // If the page is not yet generated, this will be displayed
7  // initially until getStaticProps() finishes running
8  if (router.isFallback) {
9    return <div>Loading...</div>;
10 }
11
12 // Render post...
13 }
14
15 // This function gets called at build time
16 export async function getStaticPaths() {
17   return {
18     // Only `/posts/1` and `/posts/2` are generated at build time
19     paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
20     // Enable statically generating additional pages
21     // For example: `/posts/3`
22     fallback: true,
23   };
24 }
25
26 // This also gets called at build time
27 export async function getStaticProps({ params }) {
28   // params contains the post `id`.
29   // If the route is like /posts/1, then params.id is 1
30   const res = await fetch(`https://.../posts/${params.id}`);
31   const post = await res.json();
32
33   // Pass post data to the page via props
34   return {
35     props: { post },
36     // Re-generate the post at most once per second
37     // if a request comes in
38     revalidate: 1,
39   };
40 }
41
42 export default Post;

```

Version History

Version	Changes
v13.4.0	App Router is now stable with simplified data fetching, including <code>generateStaticParams()</code>
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.

> Menu

Pages Router > ... > Functions > getStaticProps

getStaticProps

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

TS pages/index.tsx

```
1 import type { InferGetStaticPropsType, GetStaticProps } from 'next';
2
3 type Repo = {
4   name: string;
5   stargazers_count: number;
6 };
7
8 export const getStaticProps: GetStaticProps<{
9   repo: Repo;
10 }> = async () => {
11   const res = await fetch('https://api.github.com/repos/vercel/next.js');
12   const repo = await res.json();
13   return { props: { repo } };
14 };
15
16 export default function Page({
17   repo,
18 }: InferGetStaticPropsType<typeof getStaticProps>) {
19   return repo.stargazers_count;
20 }
```

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 <code>[id].js</code> , then <code>params</code> will look like <code>{ id: ... }</code> . You should use this together with <code>getStaticPaths</code> , which we'll explain later.
preview	(Deprecated for <code>draftMode</code>) <code>preview</code> is <code>true</code> if the page is in the Preview Mode and <code>false</code> otherwise.
previewData	(Deprecated for <code>draftMode</code>) The <code>preview</code> data set by <code>setPreviewData</code> .
draftMode	<code>draftMode</code> is <code>true</code> if the page is in the Draft Mode and <code>false</code> 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`.

```
1 export async function getStaticProps(context) {
2   return {
3     props: { message: `Next.js is awesome` }, // will be passed to the page component as
4   };
5 }
```

revalidate

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

```
1 // This function gets called at build time on server-side.
2 // It may be called again, on a serverless function, if
3 // revalidation is enabled and a new request comes in
```

```
4 export async function getStaticProps() {
5   const res = await fetch('https://.../posts');
6   const posts = await res.json();
7
8   return {
9     props: {
10       posts,
11     },
12     // Next.js will attempt to re-generate the page:
13     // - When a request comes in
14     // - At most once every 10 seconds
15     revalidate: 10, // In seconds
16   };
17 }
```

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](#).

```
1 export async function getStaticProps(context) {
2   const res = await fetch(`https://.../data`);
3   const data = await res.json();
4
5   if (!data) {
6     return {
7       notFound: true,
8     };
9   }
10
11   return {
12     props: { data }, // will be passed to the page component as props
13   };
14 }
```

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`.

```
1 export async function getStaticProps(context) {
2   const res = await fetch(`https://...`);
3   const data = await res.json();
4
5   if (!data) {
6     return {
7       redirect: {
8         destination: '/',
9         permanent: false,
10        // statusCode: 301
11      },
12    };
13  }
14
15  return {
16    props: { data }, // will be passed to the page component as props
17  };
18}
```

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.

```
1 import { promises as fs } from 'fs';
2 import path from 'path';
3
4 // posts will be populated at build time by getStaticProps()
5 function Blog({ posts }) {
6   return (
7     <ul>
8       {posts.map((post) => (
9         <li>
10          <h3>{post.filename}</h3>
11          <p>{post.content}</p>
12        </li>
13      ))}
14    </ul>
15  );
16}
17
18 // This function gets called at build time on server-side.
19 // It won't be called on client-side, so you can even do
20 // direct database queries.
21 export async function getStaticProps() {
22   const postsDirectory = path.join(process.cwd(), 'posts');
23   const filenames = await fs.readdir(postsDirectory);
24
25   const posts = filenames.map(async (filename) => {
26     const filePath = path.join(postsDirectory, filename);
27     const fileContents = await fs.readFile(filePath, 'utf8');
28
29     // Generally you would parse/transform the contents
30     // For example you can transform markdown to HTML here
31
32     return {
33       filename,
34       content: fileContents,
35     };
36   });
37   // By returning { props: { posts } }, the Blog component
38   // will receive `posts` as a prop at build time
39   return {
40     props: {
41       posts: await Promise.all(posts),
42     },
43   };
44 }
45
46 export default Blog;
```

Version History

Version	Changes
v13.4.0	App Router is now stable with simplified 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	<code>locale</code> , <code>locales</code> , <code>defaultLocale</code> , and <code>notFound</code> options added.
v10.0.0	<code>fallback: 'blocking'</code> return option added.
v9.5.0	Stable Incremental Static Regeneration
v9.3.0	<code>getStaticProps</code> introduced.

NextRequest and NextResponse

`next/server` provides server-only helpers for use in [Middleware](#) and [Edge API Routes](#).

NextRequest

The `NextRequest` object is an extension of the native [Request](#) ↗ interface, with the following added methods and properties:

- `cookies` - A [RequestCookies](#) ↗ instance with cookies from the `Request`. It reads/mutates the `Cookie` header of the request. See also [Using cookies in Middleware](#).
 - `get` - A method that takes a cookie `name` and returns an object with `name` and `value`. If a cookie with `name` isn't found, it returns `undefined`. If multiple cookies match, it will only return the first match.
 - `getAll` - A method that is similar to `get`, but returns a list of all the cookies with a matching `name`. If `name` is unspecified, it returns all the available cookies.
 - `set` - A method that takes an object with properties of `CookieListItem` as defined in the [W3C CookieStore API](#) ↗ spec.
 - `delete` - A method that takes either a cookie `name` or a list of names, and removes the cookies matching the name(s). Returns `true` for deleted and `false` for undeleted cookies.
 - `has` - A method that takes a cookie `name` and returns a `boolean` based on if the cookie exists (`true`) or not (`false`).
 - `clear` - A method that takes no argument and will effectively remove the `Cookie` header.
- `nextUrl` : Includes an extended, parsed, URL object that gives you access to Next.js specific properties such as `pathname`, `basePath`, `trailingSlash` and `i18n`. Includes the following properties:
 - `basePath` (`string`)
 - `buildId` (`string || undefined`)

- `defaultLocale` (`string || undefined`)
 - `domainLocale`
 - `defaultLocale` : (`string`)
 - `domain` : (`string`)
 - `http` : (`boolean || undefined`)
 - `locales` : (`string[] || undefined`)
 - `locale` (`string || undefined`)
 - `url` (`URL`)
-
- `ip` : (`string || undefined`) - Has the IP address of the `Request`. This information is provided by your hosting platform.
 - `geo` - Has the geographic location from the `Request`. This information is provided by your hosting platform. Includes the following properties:
 - `city` (`string || undefined`)
 - `country` (`string || undefined`)
 - `region` (`string || undefined`)
 - `latitude` (`string || undefined`)
 - `longitude` (`string || undefined`)

You can use the `NextRequest` object as a direct replacement for the native `Request` interface, giving you more control over how you manipulate the request.

`NextRequest` can be imported from `next/server`:

```
import type { NextRequest } from 'next/server';
```

NextFetchEvent

The `NextFetchEvent` object extends the native `FetchEvent` ↗ object, and includes the `waitFor()` ↗ method.

The `waitFor()` method can be used to prolong the execution of the function if you have other background work to make.

```
1 import { NextResponse } from 'next/server';
2 import type { NextFetchEvent, NextRequest } from 'next/server';
3
4 export function middleware(req: NextRequest, event: NextFetchEvent) {
5   event.waitUntil(
6     fetch('https://my-analytics-platform.com', {
7       method: 'POST',
8       body: JSON.stringify({ pathname: req.nextUrl.pathname }),
9     }),
10   );
11
12   return NextResponse.next();
13 }
```

The `NextFetchEvent` object can be imported from `next/server`:

```
import type { NextFetchEvent } from 'next/server';
```

NextResponse

The `NextResponse` class extends the native [Response](#) interface, with the following:

Public Methods

Public methods are available on an instance of the `NextResponse` class. Depending on your use case, you can create an instance and assign to a variable, then access the following public methods:

- `cookies` - A [ResponseCookies](#) instance with the cookies from the `Response`. It reads/mutates the `Set-Cookie` header of the response. See also [Using cookies in Middleware](#).
 - `get` - A method that takes a cookie `name` and returns an object with `name` and `value`. If a cookie with `name` isn't found, it returns `undefined`. If multiple cookies match, it will only return the first match.
 - `getAll` - A method that is similar to `get`, but returns a list of all the cookies with a matching `name`. If `name` is unspecified, it returns all the available cookies.
 - `set` - A method that takes an object with properties of `CookieListItem` as defined in the [W3C CookieStore API](#) spec.
 - `delete` - A method that takes either a cookie `name` or a list of names, and removes the cookies matching the name(s). Returns `true` for deleted and `false` for undeleted cookies.

Static Methods

The following static methods are available on the `NextResponse` class directly:

- `redirect()` - Returns a `NextResponse` with a redirect set
- `rewrite()` - Returns a `NextResponse` with a rewrite set
- `next()` - Returns a `NextResponse` that will continue the middleware chain

To use the methods above, **you must return the `NextResponse` object** returned. `NextResponse` can be imported from `next/server`:

```
import { NextResponse } from 'next/server';
```

userAgent

The `userAgent` helper allows you to interact with the user agent object from the request. It is abstracted from the native `Request` object, and is an opt in feature. It has the following properties:

- `isBot` : (`boolean`) Whether the request comes from a known bot
- `browser`
 - `name` : (`string || undefined`) The name of the browser
 - `version` : (`string || undefined`) The version of the browser, determined dynamically
- `device`
 - `model` : (`string || undefined`) The model of the device, determined dynamically
 - `type` : (`string || undefined`) The type of the browser, can be one of the following values:
`console`, `mobile`, `tablet`, `smarttv`, `wearable`, `embedded`, or `undefined`
 - `vendor` : (`string || undefined`) The vendor of the device, determined dynamically
- `engine`
 - `name` : (`string || undefined`) The name of the browser engine, could be one of the following values: `Amaya`, `Blink`, `EdgeHTML`, `Flow`, `Gecko`, `Goanna`, `iCab`, `KHTML`, `Links`, `Lynx`, `NetFront`, `NetSurf`, `Presto`, `Tasman`, `Trident`, `w3m`, `WebKit` or `undefined`
 - `version` : (`string || undefined`) The version of the browser engine, determined dynamically, or `undefined`
- `os`

- `name` : (string || undefined) The name of the OS, could be `undefined`
- `version` : (string || undefined) The version of the OS, determined dynamically, or `undefined`
- `cpu`
- `architecture` : (string || undefined) The architecture of the CPU, could be one of the following values: `68k`, `amd64`, `arm`, `arm64`, `armhf`, `avr`, `ia32`, `ia64`, `irix`, `irix64`, `mips`, `mips64`, `pa-risc`, `ppc`, `sparc`, `sparc64` or `undefined`

`userAgent` can be imported from `next/server`:

```
import { userAgent } from 'next/server';
```

```
1 import { NextRequest, NextResponse, userAgent } from 'next/server';
2
3 export function middleware(request: NextRequest) {
4   const url = request.nextUrl;
5   const { device } = userAgent(request);
6   const viewport = device.type === 'mobile' ? 'mobile' : 'desktop';
7   url.searchParams.set('viewport', viewport);
8   return NextResponse.rewrite(url);
9 }
```

FAQ

Why does `redirect` use 307 and 308?

When using `redirect()` you may notice that the status codes used are `307` for a temporary redirect, and `308` for a permanent redirect. While traditionally a `302` was used for a temporary redirect, and a `301` for a permanent redirect, many browsers changed the request method of the redirect, from a `POST` to `GET` request when using a `302`, regardless of the origins request method.

Taking the following example of a redirect from `/users` to `/people`, if you make a `POST` request to `/users` to create a new user, and are conforming to a `302` temporary redirect, the request method will be changed from a `POST` to a `GET` request. This doesn't make sense, as to create a new user, you should be making a `POST` request to `/people`, and not a `GET` request.

The introduction of the `307` status code means that the request method is preserved as `POST`.

- `302` - Temporary redirect, will change the request method from `POST` to `GET`
- `307` - Temporary redirect, will preserve the request method as `POST`

The `redirect()` method uses a `307` by default, instead of a `302` temporary redirect, meaning your requests will *always* be preserved as `POST` requests.

If you want to cause a `GET` response to a `POST` request, use `303`.

[Learn more ↗](#) about HTTP Redirects.

How do I access Environment Variables?

`process.env` can be used to access [Environment Variables](#) from Edge Middleware. They are evaluated during `next build`:

Works

```
console.log(process.env.MY_ENV_VARIABLE)
```

```
const { MY_ENV_VARIABLE } = process.env
```

```
const { "MY-ENV-VARIABLE": MY_ENV_VARIABLE } = process.env
```

Does not work

```
const getEnv = name => process.env[name]
```

> Menu

Pages Router > ... > Functions > useAmp

useAmp

► Examples

AMP support is one of our advanced features, you can [read more about AMP here](#).

To enable AMP, add the following config to your page:

```
export const config = { amp: true };
```

The `amp` config accepts the following values:

- `true` - The page will be AMP-only
- `'hybrid'` - The page will have two versions, one with AMP and another one with HTML

To learn more about the `amp` config, read the sections below.

AMP First Page

Take a look at the following example:

```
1 export const config = { amp: true };
2
3 function About(props) {
4   return <h3>My AMP About Page!</h3>;
5 }
6
7 export default About;
```

The page above is an AMP-only page, which means:

- The page has no Next.js or React client-side runtime
 - The page is automatically optimized with [AMP Optimizer](#), an optimizer that applies the same transformations as AMP caches (improves performance by up to 42%)
 - The page has a user-accessible (optimized) version of the page and a search-engine indexable (unoptimized) version of the page
-

Hybrid AMP Page

Take a look at the following example:

```
1 import { useAmp } from 'next/amp';
2
3 export const config = { amp: 'hybrid' };
4
5 function About(props) {
6   const isAmp = useAmp();
7
8   return (
9     <div>
10       <h3>My AMP About Page!</h3>
11       {isAmp ? (
12         <amp-img
13           width="300"
14           height="300"
15           src="/my-img.jpg"
16           alt="a cool image"
17           layout="responsive"
18         />
19       ) : (
20         
21       )}
22     </div>
23   );
24 }
25
26 export default About;
```

The page above is a hybrid AMP page, which means:

- The page is rendered as traditional HTML (default) and AMP HTML (by adding `?amp=1` to the URL)
- The AMP version of the page only has valid optimizations applied with AMP Optimizer so that it is indexable by search-engines

The page uses `useAmp` to differentiate between modes, it's a [React Hook ↗](#) that returns `true` if the page is using AMP, and `false` otherwise.

> Menu

Pages Router > ... > Functions > useReportWebVitals

useReportWebVitals

The `useReportWebVitals` hook allows you to report [Core Web Vitals](#), and can be used in combination with your analytics service.

JS pages/_app.js



```
1 import { useReportWebVitals } from 'next/web-vitals';
2
3 function MyApp({ Component, pageProps }) {
4   useReportWebVitals((metric) => {
5     console.log(metric);
6   });
7
8   return <Component {...pageProps} />;
9 }
```

useReportWebVitals

The `metric` object passed as the hook's argument consists of a number of properties:

- `id`: Unique identifier for the metric in the context of the current page load
- `name`: The name of the performance metric. Possible values include names of [Web Vitals](#) metrics (TTFB, FCP, LCP, FID, CLS) specific to a web application.
- `delta`: The difference between the current value and the previous value of the metric. The value is typically in milliseconds and represents the change in the metric's value over time.
- `entries`: An array of [Performance Entries](#) associated with the metric. These entries provide detailed information about the performance events related to the metric.
- `navigationType`: Indicates the [type of navigation](#) that triggered the metric collection. Possible values include "navigate", "reload", "back_forward", and "prerender".

- **rating** : A qualitative rating of the metric value, providing an assessment of the performance. Possible values are "good", "needs-improvement", and "poor". The rating is typically determined by comparing the metric value against predefined thresholds that indicate acceptable or suboptimal performance.
- **value** : The actual value or duration of the performance entry, typically in milliseconds. The value provides a quantitative measure of the performance aspect being tracked by the metric. The source of the value depends on the specific metric being measured and can come from various [Performance API](#).

Web Vitals

[Web Vitals](#) are a set of useful metrics that aim to capture the user experience of a web page. The following web vitals are all included:

- [Time to First Byte](#) (TTFB)
- [First Contentful Paint](#) (FCP)
- [Largest Contentful Paint](#) (LCP)
- [First Input Delay](#) (FID)
- [Cumulative Layout Shift](#) (CLS)
- [Interaction to Next Paint](#) (INP)

You can handle all the results of these metrics using the `name` property.

js pages/_app.js



```
1 import { useReportWebVitals } from 'next/web-vitals';
2
3 function MyApp({ Component, pageProps }) {
4   useReportWebVitals((metric) => {
5     switch (metric.name) {
6       case 'FCP': {
7         // handle FCP results
8       }
9       case 'LCP': {
10         // handle LCP results
11       }
12       // ...
13     }
14   });
15
16   return <Component {...pageProps} />;

```

Custom Metrics

In addition to the core metrics listed above, there are some additional custom metrics that measure the time it takes for the page to hydrate and render:

- `Next.js-hydration` : Length of time it takes for the page to start and finish hydrating (in ms)
- `Next.js-route-change-to-render` : Length of time it takes for a page to start rendering after a route change (in ms)
- `Next.js-render` : Length of time it takes for a page to finish render after a route change (in ms)

You can handle all the results of these metrics separately:

```
1 export function reportWebVitals(metric) {  
2   switch (metric.name) {  
3     case 'Next.js-hydration':  
4       // handle hydration results  
5       break;  
6     case 'Next.js-route-change-to-render':  
7       // handle route-change to render results  
8       break;  
9     case 'Next.js-render':  
10      // handle render results  
11      break;  
12    default:  
13      break;  
14    }  
15  }
```

These metrics work in all browsers that support the [User Timing API ↗](#).

Usage on Vercel

[Vercel Speed Insights ↗](#) are automatically configured on Vercel deployments, and don't require the use of `useReportWebVitals`. This hook is useful in local development, or if you're using a different analytics service.

Sending results to external systems

You can send results to any endpoint to measure and track real user performance on your site. For example:

```
1 useReportWebVitals((metric) => {
2   const body = JSON.stringify(metric);
3   const url = 'https://example.com/analytics';
4
5   // Use `navigator.sendBeacon()` if available, falling back to `fetch()`.
6   if (navigator.sendBeacon) {
7     navigator.sendBeacon(url, body);
8   } else {
9     fetch(url, { body, method: 'POST', keepalive: true });
10  }
11});
```

Note: If you use [Google Analytics ↗](#), using the `id` value can allow you to construct metric distributions manually (to calculate percentiles, etc.)

```
1 useReportWebVitals(metric => {
2   // Use `window.gtag` if you initialized Google Analytics as this example:
3   // https://github.com/vercel/next.js/blob/canary/examples/with-google-analytics/page.js
4   window.gtag('event', metric.name, {
5     value: Math.round(metric.name === 'CLS' ? metric.value * 1000 : metric.value), // scale CLS values
6     event_label: metric.id, // id unique to current page load
7     non_interaction: true, // avoids affecting bounce rate.
8   });
9 })
```

Read more about [sending results to Google Analytics ↗](#).

> Menu

Pages Router > ... > Functions > useRouter

useRouter

If you want to access the `router object` inside any function component in your app, you can use the `useRouter` hook, take a look at the following example:

```
1 import { useRouter } from 'next/router';
2
3 function ActiveLink({ children, href }) {
4   const router = useRouter();
5   const style = {
6     marginRight: 10,
7     color: router.asPath === href ? 'red' : 'black',
8   };
9
10  const handleClick = (e) => {
11    e.preventDefault();
12    router.push(href);
13  };
14
15  return (
16    <a href={href} onClick={handleClick} style={style}>
17      {children}
18    </a>
19  );
20}
21
22 export default ActiveLink;
```

`useRouter` is a [React Hook ↗](#), meaning it cannot be used with classes. You can either use `withRouter` or wrap your class in a function component.

router object

The following is the definition of the `router object` returned by both `useRouter` and `withRouter`:

- `pathname` : `String` - The path for current route file that comes after `/pages`. Therefore, `basePath`, `locale` and trailing slash (`trailingSlash: true`) are not included.
- `query` : `Object` - The query string parsed to an object, including [dynamic route](#) parameters. It will be an empty object during prerendering if the page doesn't use [Server-side Rendering](#). Defaults to `{}`
- `asPath` : `String` - The path as shown in the browser including the search params and respecting the `trailingSlash` configuration. `basePath` and `locale` are not included.
- `isFallback` : `boolean` - Whether the current page is in [fallback mode](#).
- `basePath` : `String` - The active `basePath` (if enabled).
- `locale` : `String` - The active locale (if enabled).
- `locales` : `String[]` - All supported locales (if enabled).
- `defaultLocale` : `String` - The current default locale (if enabled).
- `domainLocales` : `Array<{domain, defaultLocale, locales}>` - Any configured domain locales.
- `isReady` : `boolean` - Whether the router fields are updated client-side and ready for use. Should only be used inside of `useEffect` methods and not for conditionally rendering on the server. See related docs for use case with [automatically statically optimized pages](#)
- `isPreview` : `boolean` - Whether the application is currently in [preview mode](#).

Using the `asPath` field may lead to a mismatch between client and server if the page is rendered using server-side rendering or [automatic static optimization](#). Avoid using `asPath` until the `isReady` field is `true`.

The following methods are included inside `router`:

`router.push`

► Examples

Handles client-side transitions, this method is useful for cases where `next/link` is not enough.

```
router.push(url, as, options);
```

- `url` : `UrlObject | String` - The URL to navigate to (see [Node.js URL module documentation](#) ↗ for `UrlObject` properties).
- `as` : `UrlObject | String` - Optional decorator for the path that will be shown in the browser URL bar. Before Next.js 9.5.3 this was used for dynamic routes.
- `options` - Optional object with the following configuration options:

- `scroll` - Optional boolean, controls scrolling to the top of the page after navigation. Defaults to `true`
- `shallow`: Update the path of the current page without rerunning `getStaticProps`, `getServerSideProps` or `getInitialProps`. Defaults to `false`
- `locale` - Optional string, indicates locale of the new page

You don't need to use `router.push` for external URLs. `window.location` ↗ is better suited for those cases.

Navigating to `pages/about.js`, which is a predefined route:

```
1 import { useRouter } from 'next/router';
2
3 export default function Page() {
4   const router = useRouter();
5
6   return (
7     <button type="button" onClick={() => router.push('/about')}>
8       Click me
9     </button>
10  );
11 }
```

Navigating `pages/post/[pid].js`, which is a dynamic route:

```
1 import { useRouter } from 'next/router';
2
3 export default function Page() {
4   const router = useRouter();
5
6   return (
7     <button type="button" onClick={() => router.push('/post/abc')}>
8       Click me
9     </button>
10  );
11 }
```

Redirecting the user to `pages/login.js`, useful for pages behind [authentication](#):

```
1 import { useEffect } from 'react';
2 import { useRouter } from 'next/router';
3
4 // Here you would fetch and return the user
5 const useUser = () => ({ user: null, loading: false });
6
7 export default function Page() {
8   const { user, loading } = useUser();
```

```

9  const router = useRouter();
10
11 useEffect(() => {
12   if (!(user || loading)) {
13     router.push('/login');
14   }
15 }, [user, loading]);
16
17 return <p>Redirecting...</p>;
18 }

```

Resetting state after navigation

When navigating to the same page in Next.js, the page's state **will not** be reset by default as React does not unmount unless the parent component has changed.

js pages/[slug].js

```

1 import Link from 'next/link';
2 import { useState } from 'react';
3 import { useRouter } from 'next/router';
4
5 export default function Page(props) {
6   const router = useRouter();
7   const [count, setCount] = useState(0);
8   return (
9     <div>
10       <h1>Page: {router.query.slug}</h1>
11       <p>Count: {count}</p>
12       <button onClick={() => setCount(count + 1)}>Increase count</button>
13       <Link href="/one">one</Link> <Link href="/two">two</Link>
14     </div>
15   );
16 }

```

In the above example, navigating between `/one` and `/two` **will not** reset the count. The `useState` is maintained between renders because the top-level React component, `Page`, is the same.

If you do not want this behavior, you have a couple of options:

- Manually ensure each state is updated using `useEffect`. In the above example, that could look like:

```

1 useEffect(() => {
2   setCount(0);
3 }, [router.query.slug]);

```

- Use a React `key` to tell React to remount the component [↗](#). To do this for all pages, you can use a custom app:



```

1 import { useRouter } from 'next/router';
2
3 export default function MyApp({ Component, pageProps }) {
4   const router = useRouter();
5   return <Component key={router.asPath} {...pageProps} />;
6 }

```

With URL object

You can use a URL object in the same way you can use it for `next/link`. Works for both the `url` and `as` parameters:

```

1 import { useRouter } from 'next/router';
2
3 export default function ReadMore({ post }) {
4   const router = useRouter();
5
6   return (
7     <button
8       type="button"
9       onClick={() => {
10         router.push({
11           pathname: '/post/[pid]',
12           query: { pid: post.id },
13         });
14       }}
15     >
16       Click here to read more
17     </button>
18   );
19 }

```

router.replace

Similar to the `replace` prop in `next/link`, `router.replace` will prevent adding a new URL entry into the `history` stack.

```
router.replace(url, as, options);
```

- The API for `router.replace` is exactly the same as the API for `router.push`.

Take a look at the following example:

```
1 import { useRouter } from 'next/router';
```

```
2
3 export default function Page() {
4   const router = useRouter();
5
6   return (
7     <button type="button" onClick={() => router.replace('/home')}>
8       Click me
9     </button>
10  );
11}
```

router.prefetch

Prefetch pages for faster client-side transitions. This method is only useful for navigations without `next/link`, as `next/link` takes care of prefetching pages automatically.

This is a production only feature. Next.js doesn't prefetch pages in development.

```
router.prefetch(url, as, options);
```

- `url` - The URL to prefetch, including explicit routes (e.g. `/dashboard`) and dynamic routes (e.g. `/product/[id]`)
- `as` - Optional decorator for `url`. Before Next.js 9.5.3 this was used to prefetch dynamic routes.
- `options` - Optional object with the following allowed fields:
 - `locale` - allows providing a different locale from the active one. If `false`, `url` has to include the locale as the active locale won't be used.

Let's say you have a login page, and after a login, you redirect the user to the dashboard. For that case, we can prefetch the dashboard to make a faster transition, like in the following example:

```
1 import { useCallback, useEffect } from 'react';
2 import { useRouter } from 'next/router';
3
4 export default function Login() {
5   const router = useRouter();
6   const handleSubmit = useCallback((e) => {
7     e.preventDefault();
8
9     fetch('/api/login', {
10       method: 'POST',
11       headers: { 'Content-Type': 'application/json' },
12       body: JSON.stringify({
13         /* Form data */
14       }),
15     })
16     .then((res) => res.json())
17     .then(({ token }) => {
18       if (token) {
19         router.prefetch('/dashboard', { locale: false });
20         router.push('/dashboard');
21       }
22     });
23   });
24
25   return (
26     <form onSubmit={handleSubmit}>
27       <input type="text" name="email" placeholder="Email" />
28       <input type="password" name="password" placeholder="Password" />
29       <button type="submit">Login</button>
30     </form>
31   );
32}
```

```
15     }).then((res) => {
16         // Do a fast client-side transition to the already prefetched dashboard page
17         if (res.ok) router.push('/dashboard');
18     });
19 }, []);
20
21 useEffect(() => {
22     // Prefetch the dashboard page
23     router.prefetch('/dashboard');
24 }, [router]);
25
26 return (
27     <form onSubmit={handleSubmit}>
28         {/* Form fields */}
29         <button type="submit">Login</button>
30     </form>
31 );
32 }
```

router.beforePopState

In some cases (for example, if using a [Custom Server](#)), you may wish to listen to [popstate](#) and do something before the router acts on it.

```
router.beforePopState(cb);
```

- `cb` - The function to run on incoming `popstate` events. The function receives the state of the event as an object with the following props:
 - `url: String` - the route for the new state. This is usually the name of a `page`
 - `as: String` - the url that will be shown in the browser
 - `options: Object` - Additional options sent by `router.push`

If `cb` returns `false`, the Next.js router will not handle `popstate`, and you'll be responsible for handling it in that case. See [Disabling file-system routing](#).

You could use `beforePopState` to manipulate the request, or force a SSR refresh, as in the following example:

```
1 import { useEffect } from 'react';
2 import { useRouter } from 'next/router';
3
4 export default function Page() {
5     const router = useRouter();
6
7     useEffect(() => {
8         router.beforePopState(({ url, as, options }) => {
```

```
9     // I only want to allow these two routes!
10    if (as !== '/' && as !== '/other') {
11        // Have SSR render bad routes as a 404.
12        window.location.href = as;
13        return false;
14    }
15
16    return true;
17 });
18 }, [router]);
19
20 return <p>Welcome to the page</p>;
21 }
```

router.back

Navigate back in history. Equivalent to clicking the browser's back button. It executes

```
window.history.back()
```

```
1 import { useRouter } from 'next/router';
2
3 export default function Page() {
4     const router = useRouter();
5
6     return (
7         <button type="button" onClick={() => router.back()}>
8             Click here to go back
9         </button>
10    );
11 }
```

router.reload

Reload the current URL. Equivalent to clicking the browser's refresh button. It executes

```
window.location.reload()
```

```
1 import { useRouter } from 'next/router';
2
3 export default function Page() {
4     const router = useRouter();
5
6     return (
7         <button type="button" onClick={() => router.reload()}>
8             Click here to reload
9         </button>
10    );
11 }
```

router.events

► Examples

You can listen to different events happening inside the Next.js Router. Here's a list of supported events:

- `routeChangeStart(url, { shallow })` - Fires when a route starts to change
- `routeChangeComplete(url, { shallow })` - Fires when a route changed completely
- `routeChangeError(err, url, { shallow })` - Fires when there's an error when changing routes, or a route load is cancelled
 - `err.cancelled` - Indicates if the navigation was cancelled
- `beforeHistoryChange(url, { shallow })` - Fires before changing the browser's history
- `hashChangeStart(url, { shallow })` - Fires when the hash will change but not the page
- `hashChangeComplete(url, { shallow })` - Fires when the hash has changed but not the page

Note: Here `url` is the URL shown in the browser, including the `basePath`.

For example, to listen to the router event `routeChangeStart`, open or create `pages/_app.js` and subscribe to the event, like so:

```
1 import { useEffect } from 'react';
2 import { useRouter } from 'next/router';
3
4 export default function MyApp({ Component, pageProps }) {
5   const router = useRouter();
6
7   useEffect(() => {
8     const handleRouteChange = (url, { shallow }) => {
9       console.log(
10         `App is changing to ${url} ${
11           shallow ? 'with' : 'without'
12           } shallow routing`,
13       );
14     };
15
16     router.events.on('routeChangeStart', handleRouteChange);
17
18     // If the component is unmounted, unsubscribe
19     // from the event with the `off` method:
20     return () => {
21       router.events.off('routeChangeStart', handleRouteChange);
22     };
23   }, [router]);
24
25   return <Component {...pageProps} />;
}
```

We use a [Custom App](#) (`pages/_app.js`) for this example to subscribe to the event because it's not unmounted on page navigations, but you can subscribe to router events on any component in your application.

Router events should be registered when a component mounts ([useEffect](#) ↗ or [componentDidMount](#) ↗ / [componentWillUnmount](#) ↗) or imperatively when an event happens.

If a route load is cancelled (for example, by clicking two links rapidly in succession), `routeChangeError` will fire. And the passed `err` will contain a `cancelled` property set to `true`, as in the following example:

```

1 import { useEffect } from 'react';
2 import { useRouter } from 'next/router';
3
4 export default function MyApp({ Component, pageProps }) {
5   const router = useRouter();
6
7   useEffect(() => {
8     const handleRouteChangeError = (err, url) => {
9       if (err.cancelled) {
10         console.log(`Route to ${url} was cancelled!`);
11       }
12     };
13
14     router.events.on('routeChangeError', handleRouteChangeError);
15
16     // If the component is unmounted, unsubscribe
17     // from the event with the `off` method:
18     return () => {
19       router.events.off('routeChangeError', handleRouteChangeError);
20     };
21   }, [router]);
22
23   return <Component {...pageProps} />;
24 }
```

Potential ESLint errors

Certain methods accessible on the `router` object return a Promise. If you have the ESLint rule, [no-floating-promises](#) ↗ enabled, consider disabling it either globally, or for the affected line.

If your application needs this rule, you should either `void` the promise – or use an `async` function, `await` the Promise, then void the function call. **This is not applicable when the method is called from inside an**

onClick handler.

The affected methods are:

- `router.push`
- `router.replace`
- `router.prefetch`

Potential solutions

```
1 import { useEffect } from 'react';
2 import { useRouter } from 'next/router';
3
4 // Here you would fetch and return the user
5 const useUser = () => ({ user: null, loading: false });
6
7 export default function Page() {
8   const { user, loading } = useUser();
9   const router = useRouter();
10
11   useEffect(() => {
12     // disable the linting on the next line - This is the cleanest solution
13     // eslint-disable-next-line no-floating-promises
14     router.push('/login');
15
16     // void the Promise returned by router.push
17     if (!(user || loading)) {
18       void router.push('/login');
19     }
20     // or use an async function, await the Promise, then void the function call
21     async function handleRouteChange() {
22       if (!(user || loading)) {
23         await router.push('/login');
24       }
25     }
26     void handleRouteChange();
27   }, [user, loading]);
28
29   return <p>Redirecting...</p>;
30 }
```

withRouter

If `useRouter` is not the best fit for you, `withRouter` can also add the same `router object` to any component.

Usage

```
1 import { withRouter } from 'next/router';
2
3 function Page({ router }) {
4   return <p>{router.pathname}</p>;
5 }
6
7 export default withRouter(Page);
```

TypeScript

To use class components with `withRouter`, the component needs to accept a router prop:

```
1 import React from 'react';
2 import { withRouter, NextRouter } from 'next/router';
3
4 interface WithRouterProps {
5   router: NextRouter;
6 }
7
8 interface MyComponentProps extends WithRouterProps {}
9
10 class MyComponent extends React.Component<MyComponentProps> {
11   render() {
12     return <p>{this.props.router.pathname}</p>;
13   }
14 }
15
16 export default withRouter(MyComponent);
```

> Menu

Pages Router > API Reference > Next.js CLI

Next.js CLI

The Next.js CLI allows you to start, build, and export your application.

To get a list of the available CLI commands, run the following command inside your project directory:

>_ Terminal

```
npx next -h
```

(*npx*[↗] comes with npm 5.2+ and higher)

The output should look like this:

>_ Terminal

```
1 Usage
2   $ next <command>
3
4 Available commands
5   build, start, export, dev, lint, telemetry, info
6
7 Options
8   --version, -v    Version number
9   --help, -h       Displays this message
10
11 For more information run a command with the --help flag
12   $ next build --help
```

You can pass any [node arguments](#)[↗] to `next` commands:

>_ Terminal

```
1 NODE_OPTIONS='--throw-deprecation' next
2 NODE_OPTIONS='-r esm' next
3 NODE_OPTIONS='--inspect' next
```

Note: Running `next` without a command is the same as running `next dev`

Build

`next build` creates an optimized production build of your application. The output displays information about each route.

- **Size** – The number of assets downloaded when navigating to the page client-side. The size for each route only includes its dependencies.
- **First Load JS** – The number of assets downloaded when visiting the page from the server. The amount of JS shared by all is shown as a separate metric.

Both of these values are **compressed with gzip**. The first load is indicated by green, yellow, or red. Aim for green for performant applications.

You can enable production profiling for React with the `--profile` flag in `next build`. This requires Next.js 9.5[↗]:

> Terminal

```
next build --profile
```

After that, you can use the profiler in the same way as you would in development.

You can enable more verbose build output with the `--debug` flag in `next build`. This requires Next.js 9.5.3:

> Terminal

```
next build --debug
```

With this flag enabled additional build output like rewrites, redirects, and headers will be shown.

Development

`next dev` starts the application in development mode with hot-code reloading, error reporting, and more:

The application will start at `http://localhost:3000` by default. The default port can be changed with `-p`, like so:

```
>_ Terminal
```

```
npx next dev -p 4000
```

Or using the `PORT` environment variable:

```
>_ Terminal
```

```
PORT=4000 npx next dev
```

Note: `PORT` cannot be set in `.env` as booting up the HTTP server happens before any other code is initialized.

You can also set the hostname to be different from the default of `0.0.0.0`, this can be useful for making the application available for other devices on the network. The default hostname can be changed with `-H`, like so:

```
>_ Terminal
```

```
npx next dev -H 192.168.1.2
```

Production

`next start` starts the application in production mode. The application should be compiled with `next build` first.

The application will start at `http://localhost:3000` by default. The default port can be changed with `-p`, like so:

>_ Terminal



```
npx next start -p 4000
```

Or using the `PORT` environment variable:

>_ Terminal



```
PORT=4000 npx next start
```

Note: `PORT` cannot be set in `.env` as booting up the HTTP server happens before any other code is initialized.

Note: `next start` cannot be used with `output: 'standalone'` or `output: 'export'`.

Keep Alive Timeout

When deploying Next.js behind a downstream proxy (e.g. a load-balancer like AWS ELB/ALB) it's important to configure Next's underlying HTTP server with [keep-alive timeouts](#) ↗ that are *larger* than the downstream proxy's timeouts. Otherwise, once a keep-alive timeout is reached for a given TCP connection, Node.js will immediately terminate that connection without notifying the downstream proxy. This results in a proxy error whenever it attempts to reuse a connection that Node.js has already terminated.

To configure the timeout values for the production Next.js server, pass `--keepAliveTimeout` (in milliseconds) to `next start`, like so:

>_ Terminal



```
npx next start --keepAliveTimeout 70000
```

Lint

`next lint` runs ESLint for all files in the `pages/`, `app` (only if the experimental `appDir` feature is enabled), `components/`, `lib/`, and `src/` directories. It also provides a guided setup to install any required dependencies if ESLint is not already configured in your application.

If you have other directories that you would like to lint, you can specify them using the `--dir` flag:

>_ Terminal



```
next lint --dir utils
```

Telemetry

Next.js collects **completely anonymous** telemetry data about general usage. Participation in this anonymous program is optional, and you may opt-out if you'd not like to share any information.

To learn more about Telemetry, [please read this document ↗](#).

Next Info

`next info` prints relevant details about the current system which can be used to report Next.js bugs. This information includes Operating System platform/arch/version, Binaries (Node.js, npm, Yarn, pnpm) and npm package versions (`next`, `react`, `react-dom`).

Running the following in your project's root directory:

>_ Terminal



```
next info
```

will give you information like this example:

>_ Terminal



```
1
2  Operating System:
3    Platform: linux
4    Arch: x64
5    Version: #22-Ubuntu SMP Fri Nov 5 13:21:36 UTC 2021
6  Binaries:
7    Node: 16.13.0
8    npm: 8.1.0
9    Yarn: 1.22.17
10   pnpm: 6.24.2
11  Relevant packages:
```

```
12      next: 12.0.8
13      react: 17.0.2
14      react-dom: 17.0.2
15
```

This information should then be pasted into GitHub Issues.

> Menu

Pages Router > API Reference > next.config.js Options

next.config.js Options

Next.js can be configured through a `next.config.js` file in the root of your project directory.

JS `next.config.js`

```
1  /** @type {import('next').NextConfig} */
2  const nextConfig = {
3    /* config options here */
4  };
5
6 module.exports = nextConfig;
```



This page documents all the available configuration options:

assetPrefix

Learn how to use the `assetPrefix` config option to configure your CDN.

basePath

Use `'basePath'` to deploy a Next.js application under a sub-path of a domain.

compress

Next.js provides gzip compression to compress rendered content and static files, it only works with the `server` target. Learn more about it [here](#).

devIndicators

Optimized pages include an indicator to let you know if it's being statically optimized. You can opt-out of it here.

[distDir](#)

Set a custom build directory to use instead of the default .next directory.

[env](#)

Learn to add and access environment variables in your Next.js application at build time.

[eslint](#)

Next.js reports ESLint errors and warnings during builds by default. Learn how to opt-out of this behavior here.

[exportPathMap](#)

Customize the pages that will be exported as HTML files when using `next export`.

[generateBuildId](#)

Configure the build id, which is used to identify the current build in which your application is being served.

[generateEtags](#)

Next.js will generate etags for every page by default. Learn more about how to disable etag generation here.

[headers](#)

Add custom HTTP headers to your Next.js app.

[httpAgentOptions](#)

Next.js will automatically use HTTP Keep-Alive by default. Learn more about how to disable HTTP Keep-Alive here.

images

Custom configuration for the next/image loader

onDemandEntries

Configure how Next.js will dispose and keep in memory pages created in development.

output

Next.js automatically traces which files are needed by each page to allow for easy deployment of your application. Learn how it works here.

pageExtensions

Extend the default page extensions used by Next.js when resolving pages in the Pages Router.

poweredByHeader

Next.js will add the `x-powered-by` header by default. Learn to opt-out of it here.

productionBrowserSourceMaps

Enables browser source map generation during the production build.

reactStrictMode

The complete Next.js runtime is now Strict Mode-compliant, learn how to opt-in

redirects

Add redirects to your Next.js app.

rewrites

Add rewrites to your Next.js app.

Runtime Config

Add client and server runtime configuration to your Next.js app.

trailingSlash

Configure Next.js pages to resolve with or without a trailing slash.

transpilePackages

Automatically transpile and bundle dependencies from local packages (like monorepos) or from external dependencies (`node_modules`).

turbo

Configure Next.js with Turbopack-specific options

typescript

Next.js reports TypeScript errors by default. Learn to opt-out of this behavior here.

urlImports

Configure Next.js to allow importing modules from external URLs (experimental).

webpack

Learn how to customize the webpack config used by Next.js

webVitalsAttribution

Learn how to use the webVitalsAttribution option to pinpoint the source of Web Vitals issues.

> Menu

Pages Router > ... > next.config.js Options > assetPrefix

assetPrefix

Attention: Deploying to Vercel automatically configures a global CDN for your Next.js project. You do not need to manually setup an Asset Prefix.

Note: Next.js 9.5+ added support for a customizable [Base Path](#), which is better suited for hosting your application on a sub-path like `/docs`. We do not suggest you use a custom Asset Prefix for this use case.

To set up a [CDN](#), you can set up an asset prefix and configure your CDN's origin to resolve to the domain that Next.js is hosted on.

Open `next.config.js` and add the `assetPrefix` config:

JS next.config.js

```
1 const isProd = process.env.NODE_ENV === 'production';
2
3 module.exports = {
4   // Use the CDN in production and localhost for development.
5   assetPrefix: isProd ? 'https://cdn.mydomain.com' : undefined,
6 }
```

Next.js will automatically use your asset prefix for the JavaScript and CSS files it loads from the `/_next/` path (`.next/static/` folder). For example, with the above configuration, the following request for a JS chunk:

```
/_next/static/chunks/4b9b41aaa062cbbfeff4add70f256968c51ece5d.4d708494b3aed70c04f0.js
```

Would instead become:

```
https://cdn.mydomain.com/_next/static/chunks/4b9b41aaa062cbbfeff4add70f256968c51ece5d.4d708494b
```

The exact configuration for uploading your files to a given CDN will depend on your CDN of choice. The only folder you need to host on your CDN is the contents of `.next/static/`, which should be uploaded as `_next/static/` as the above URL request indicates. **Do not upload the rest of your `.next/` folder**, as you should not expose your server code and other configuration to the public.

While `assetPrefix` covers requests to `_next/static`, it does not influence the following paths:

- Files in the `public` folder; if you want to serve those assets over a CDN, you'll have to introduce the prefix yourself
- `/_next/data/` requests for `getServerSideProps` pages. These requests will always be made against the main domain since they're not static.
- `/_next/data/` requests for `getStaticProps` pages. These requests will always be made against the main domain to support [Incremental Static Generation](#), even if you're not using it (for consistency).

> Menu

Pages Router > ... > next.config.js Options > basePath

basePath

To deploy a Next.js application under a sub-path of a domain you can use the `basePath` config option.

`basePath` allows you to set a path prefix for the application. For example, to use `/docs` instead of `''` (an empty string, the default), open `next.config.js` and add the `basePath` config:

js next.config.js

```
1 module.exports = {  
2   basePath: '/docs',  
3 };
```

Note: This value must be set at build time and cannot be changed without re-building as the value is inlined in the client-side bundles.

Links

When linking to other pages using `next/link` and `next/router` the `basePath` will be automatically applied.

For example, using `/about` will automatically become `/docs/about` when `basePath` is set to `/docs`.

```
1 export default function HomePage() {  
2   return (  
3     <>  
4       <Link href="/about">About Page</Link>  
5     </>  
6   );  
7 }
```

Output html:

```
<a href="/docs/about">About Page</a>
```

This makes sure that you don't have to change all links in your application when changing the `basePath` value.

Images

When using the `next/image` component, you will need to add the `basePath` in front of `src`.

For example, using `/docs/me.png` will properly serve your image when `basePath` is set to `/docs`.

```
1 import Image from 'next/image';
2
3 function Home() {
4   return (
5     <>
6       <h1>My Homepage</h1>
7       <Image
8         src="/docs/me.png"
9         alt="Picture of the author"
10        width={500}
11        height={500}
12      />
13      <p>Welcome to my homepage!</p>
14    </>
15  );
16}
17
18 export default Home;
```

> Menu

Pages Router > ... > next.config.js Options > compress

compress

Next.js provides [gzip](#) compression to compress rendered content and static files. In general you will want to enable compression on a HTTP proxy like [nginx](#), to offload load from the `Node.js` process.

To disable **compression**, open `next.config.js` and disable the `compress` config:

`js next.config.js`

```
1 module.exports = {
2   compress: false,
3 };
```


> Menu

Pages Router > ... > next.config.js Options > devIndicators

devIndicators

Note: This indicator was removed in Next.js version 10.0.1. We recommend upgrading to the latest version of Next.js.

When a page qualifies for [Automatic Static Optimization](#) we show an indicator to let you know.

This is helpful since automatic static optimization can be very beneficial and knowing immediately in development if the page qualifies can be useful.

In some cases this indicator might not be useful, like when working on electron applications. To remove it open `next.config.js` and disable the `autoPrerender` config in `devIndicators`:

JS next.config.js



```
1 module.exports = {
2   devIndicators: {
3     autoPrerender: false,
4   },
5 };
```


> Menu

Pages Router > ... > next.config.js Options > distDir

distDir

You can specify a name to use for a custom build directory to use instead of `.next`.

Open `next.config.js` and add the `distDir` config:

`js` `next.config.js`

```
1 module.exports = {  
2   distDir: 'build',  
3 };
```



Now if you run `next build` Next.js will use `build` instead of the default `.next` folder.

`distDir` **should not** leave your project directory. For example, `../build` is an **invalid** directory.

> Menu

Pages Router > ... > next.config.js Options > env

env

Since the release of [Next.js 9.4](#) we now have a more intuitive and ergonomic experience for [adding environment variables](#). Give it a try!

► Examples

Note: environment variables specified in this way will **always** be included in the JavaScript bundle, prefixing the environment variable name with `NEXT_PUBLIC_` only has an effect when specifying them [through the environment or .env files](#).

To add environment variables to the JavaScript bundle, open `next.config.js` and add the `env` config:

js next.config.js

```
1 module.exports = {
2   env: {
3     customKey: 'my-value',
4   },
5 };
```



Now you can access `process.env.customKey` in your code. For example:

```
1 function Page() {
2   return <h1>The value of customKey is: {process.env.customKey}</h1>;
3 }
4
5 export default Page;
```

Next.js will replace `process.env.customKey` with `'my-value'` at build time. Trying to destructure `process.env` variables won't work due to the nature of webpack [DefinePlugin](#).

For example, the following line:

```
return <h1>The value of customKey is: {process.env.customKey}</h1>;
```

Will end up being:

```
return <h1>The value of customKey is: {'my-value'}</h1>;
```

> Menu

Pages Router > ... > next.config.js Options > eslint

eslint

When ESLint is detected in your project, Next.js fails your **production build** (`next build`) when errors are present.

If you'd like Next.js to produce production code even when your application has ESLint errors, you can disable the built-in linting step completely. This is not recommended unless you already have ESLint configured to run in a separate part of your workflow (for example, in CI or a pre-commit hook).

Open `next.config.js` and enable the `ignoreDuringBuilds` option in the `eslint` config:

JS next.config.js



```
1 module.exports = {
2   eslint: {
3     // Warning: This allows production builds to successfully complete even if
4     // your project has ESLint errors.
5     ignoreDuringBuilds: true,
6   },
7 };
```


> Menu

Pages Router > ... > next.config.js Options > exportPathMap

exportPathMap

This feature is exclusive to `next export` and currently **deprecated** in favor of `getStaticPaths` with `pages` or `generateStaticParams` with `app`.

► Examples

`exportPathMap` allows you to specify a mapping of request paths to page destinations, to be used during export. Paths defined in `exportPathMap` will also be available when using `next dev`.

Let's start with an example, to create a custom `exportPathMap` for an app with the following pages:

- `pages/index.js`
- `pages/about.js`
- `pages/post.js`

Open `next.config.js` and add the following `exportPathMap` config:

`js` `next.config.js`

```
1 module.exports = {
2   exportPathMap: async function (
3     defaultPathMap,
4     { dev, dir, outDir, distDir, buildId },
5   ) {
6     return {
7       '/': { page: '/' },
8       '/about': { page: '/about' },
9       '/p/hello-nextjs': { page: '/post', query: { title: 'hello-nextjs' } },
10      '/p/learn-nextjs': { page: '/post', query: { title: 'learn-nextjs' } },
11      '/p/deploy-nextjs': { page: '/post', query: { title: 'deploy-nextjs' } },
12    };
13  },
14};
```

Note: the `query` field in `exportPathMap` cannot be used with automatically statically optimized pages or `getStaticProps` pages as they are rendered to HTML files at build-time and additional query information cannot be provided during `next export`.

The pages will then be exported as HTML files, for example, `/about` will become `/about.html`.

`exportPathMap` is an `async` function that receives 2 arguments: the first one is `defaultPathMap`, which is the default map used by Next.js. The second argument is an object with:

- `dev` - `true` when `exportPathMap` is being called in development. `false` when running `next export`. In development `exportPathMap` is used to define routes.
- `dir` - Absolute path to the project directory
- `outDir` - Absolute path to the `out/` directory (configurable with `-o`). When `dev` is `true` the value of `outDir` will be `null`.
- `distDir` - Absolute path to the `.next/` directory (configurable with the `distDir` config)
- `buildId` - The generated build id

The returned object is a map of pages where the `key` is the `pathname` and the `value` is an object that accepts the following fields:

- `page : String` - the page inside the `pages` directory to render
- `query : Object` - the `query` object passed to `getInitialProps` when prerendering. Defaults to `{}`

The exported `pathname` can also be a filename (for example, `/readme.md`), but you may need to set the `Content-Type` header to `text/html` when serving its content if it is different than `.html`.

Adding a trailing slash

It is possible to configure Next.js to export pages as `index.html` files and require trailing slashes, `/about` becomes `/about/index.html` and is routable via `/about/`. This was the default behavior prior to Next.js 9.

To switch back and add a trailing slash, open `next.config.js` and enable the `trailingSlash` config:

`JS` `next.config.js`

```
1 module.exports = {
```

```
2   trailingSlash: true,  
3 };
```

Customizing the output directory

`next export` will use `out` as the default output directory, you can customize this using the `-o` argument, like so:

>_ Terminal

```
next export -o outdir
```

Warning: Using `exportPathMap` is deprecated and is overridden by `getStaticPaths` inside `pages`. We don't recommend using them together.

> Menu

Pages Router > ... > next.config.js Options > generateBuildId

generateBuildId

Next.js uses a constant id generated at build time to identify which version of your application is being served. This can cause problems in multi-server deployments when `next build` is run on every server. In order to keep a consistent build id between builds you can provide your own build id.

Open `next.config.js` and add the `generateBuildId` function:

js next.config.js



```
1 module.exports = {
2   generateBuildId: async () => {
3     // You can, for example, get the latest git commit hash here
4     return 'my-build-id';
5   },
6 };
```


> Menu

Pages Router > ... > next.config.js Options > generateEtags

generateEtags

Next.js will generate [etags](#) for every page by default. You may want to disable etag generation for HTML pages depending on your cache strategy.

Open `next.config.js` and disable the `generateEtags` option:

JS next.config.js



```
1 module.exports = {  
2   generateEtags: false,  
3 };
```


> Menu

Pages Router > ... > next.config.js Options > headers

headers

Headers allow you to set custom HTTP headers on the response to an incoming request on a given path.

To set custom HTTP headers you can use the `headers` key in `next.config.js`:

js next.config.js

```
1 module.exports = {
2   async headers() {
3     return [
4       {
5         source: '/about',
6         headers: [
7           {
8             key: 'x-custom-header',
9             value: 'my custom header value',
10            },
11            {
12              key: 'x-another-custom-header',
13              value: 'my other custom header value',
14            },
15          ],
16        },
17      ];
18    },
19  };
```

`headers` is an `async` function that expects an array to be returned holding objects with `source` and `headers` properties:

- `source` is the incoming request path pattern.
- `headers` is an array of response header objects, with `key` and `value` properties.
- `basePath: false` or `undefined` - if `false` the `basePath` won't be included when matching, can be used for external rewrites only.
- `locale: false` or `undefined` - whether the `locale` should not be included when matching.

- `has` is an array of `has objects` with the `type`, `key` and `value` properties.
- `missing` is an array of `missing objects` with the `type`, `key` and `value` properties.

Headers are checked before the filesystem which includes pages and `/public` files.

Header Overriding Behavior

If two headers match the same path and set the same header key, the last header key will override the first. Using the below headers, the path `/hello` will result in the header `x-hello` being `world` due to the last header value set being `world`.

js next.config.js

```
1 module.exports = {
2   async headers() {
3     return [
4       {
5         source: '/:path*',
6         headers: [
7           {
8             key: 'x-hello',
9             value: 'there',
10            },
11          ],
12        },
13        {
14          source: '/hello',
15          headers: [
16            {
17              key: 'x-hello',
18              value: 'world',
19            },
20          ],
21        },
22      ];
23    },
24  };
}
```



Path Matching

Path matches are allowed, for example `/blog/:slug` will match `/blog/hello-world` (no nested paths):



```
1 module.exports = {
2   async headers() {
3     return [
4       {
5         source: '/blog/:slug',
6         headers: [
7           {
8             key: 'x-slug',
9             value: ':slug', // Matched parameters can be used in the value
10            },
11            {
12              key: 'x-slug-:slug', // Matched parameters can be used in the key
13              value: 'my other custom header value',
14            },
15            ],
16          },
17        ];
18      },
19    };
```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:



```
1 module.exports = {
2   async headers() {
3     return [
4       {
5         source: '/blog/:slug*',
6         headers: [
7           {
8             key: 'x-slug',
9             value: ':slug*', // Matched parameters can be used in the value
10            },
11            {
12              key: 'x-slug-:slug*', // Matched parameters can be used in the key
13              value: 'my other custom header value',
14            },
15            ],
16          },
17        ];
18      },
19    };
```

Regex Path Matching

To match a regex path you can wrap the regex in parenthesis after a parameter, for example

`/blog/:slug(\d{1,})` will match `/blog/123` but not `/blog/abc`:

js next.config.js

```
1 module.exports = {
2   async headers() {
3     return [
4       {
5         source: '/blog/:post(\d{1,})',
6         headers: [
7           {
8             key: 'x-post',
9             value: ':post',
10            },
11          ],
12        },
13      ];
14    },
15  };
```

The following characters `(`, `)`, `{`, `}`, `:`, `*`, `+`, `?` are used for regex path matching, so when used in the `source` as non-special values they must be escaped by adding `\` before them:

js next.config.js

```
1 module.exports = {
2   async headers() {
3     return [
4       {
5         // this will match `/english(default)/something` being requested
6         source: '/english\\(default\\)\\/:slug',
7         headers: [
8           {
9             key: 'x-header',
10             value: 'value',
11           },
12         ],
13       },
14     ];
15   },
16 };
```

Header, Cookie, and Query Matching

To only apply a header when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the header to be applied.

`has` and `missing` items can have the following fields:

- `type : String` - must be either `header`, `cookie`, `host`, or `query`.
- `key : String` - the key from the selected type to match against.
- `value : String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

js next.config.js

```
1 module.exports = {
2   async headers() {
3     return [
4       // if the header `x-add-header` is present,
5       // the `x-another-header` header will be applied
6       {
7         source: '/:path*',
8         has: [
9           {
10             type: 'header',
11             key: 'x-add-header',
12           },
13         ],
14         headers: [
15           {
16             key: 'x-another-header',
17             value: 'hello',
18           },
19         ],
20       },
21       // if the header `x-no-header` is not present,
22       // the `x-another-header` header will be applied
23       {
24         source: '/:path*',
25         missing: [
26           {
27             type: 'header',
28             key: 'x-no-header',
29           },
30         ],
31         headers: [
```

```
32      {
33          key: 'x-another-header',
34          value: 'hello',
35      },
36  ],
37 },
38 // if the source, query, and cookie are matched,
39 // the `x-authorized` header will be applied
40 {
41     source: '/specific/:path*',
42     has: [
43         {
44             type: 'query',
45             key: 'page',
46             // the page value will not be available in the
47             // header key/values since value is provided and
48             // doesn't use a named capture group e.g. (?<page>home)
49             value: 'home',
50         },
51         {
52             type: 'cookie',
53             key: 'authorized',
54             value: 'true',
55         },
56     ],
57     headers: [
58         {
59             key: 'x-authorized',
60             value: ':authorized',
61         },
62     ],
63 },
64 // if the header `x-authorized` is present and
65 // contains a matching value, the `x-another-header` will be applied
66 {
67     source: '/:path*',
68     has: [
69         {
70             type: 'header',
71             key: 'x-authorized',
72             value: '(?<authorized>yes|true)',
73         },
74     ],
75     headers: [
76         {
77             key: 'x-another-header',
78             value: ':authorized',
79         },
80     ],
81 },
82 // if the host is `example.com`,
83 // this header will be applied
84 {
85     source: '/:path*',
86     has: [
87         {
```

```
88         type: 'host',
89         value: 'example.com',
90     },
91 ],
92 headers: [
93     {
94         key: 'x-another-header',
95         value: ':authorized',
96     },
97 ],
98 },
99 ],
100 },
101 };
```

Headers with basePath support

When leveraging `basePath` support with headers each `source` is automatically prefixed with the `basePath` unless you add `basePath: false` to the header:

js next.config.js

```
1 module.exports = {
2   basePath: '/docs',
3
4   async headers() {
5     return [
6       {
7         source: '/with-basePath', // becomes /docs/with-basePath
8         headers: [
9           {
10             key: 'x-hello',
11             value: 'world',
12           },
13         ],
14       },
15       {
16         source: '/without-basePath', // is not modified since basePath: false is set
17         headers: [
18           {
19             key: 'x-hello',
20             value: 'world',
21           },
22         ],
23         basePath: false,
24       },
25     ];
26   },
27 }
```

```
27 };
```

Headers with i18n support

When leveraging [i18n support](#) with headers each `source` is automatically prefixed to handle the configured `locales` unless you add `locale: false` to the header. If `locale: false` is used you must prefix the `source` with a locale for it to be matched correctly.

JS next.config.js

```
1 module.exports = {
2   i18n: {
3     locales: ['en', 'fr', 'de'],
4     defaultLocale: 'en',
5   },
6
7   async headers() {
8     return [
9       {
10         source: '/with-locale', // automatically handles all locales
11         headers: [
12           {
13             key: 'x-hello',
14             value: 'world',
15           },
16         ],
17       },
18       {
19         // does not handle locales automatically since locale: false is set
20         source: '/nl/with-locale-manual',
21         locale: false,
22         headers: [
23           {
24             key: 'x-hello',
25             value: 'world',
26           },
27         ],
28       },
29       {
30         // this matches '/' since `en` is the defaultLocale
31         source: '/en',
32         locale: false,
33         headers: [
34           {
35             key: 'x-hello',
36             value: 'world',
37           },
38         ],
39       }
40     ]
41   }
42 }
```

```
39 },
40 {
41     // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
42     // `/` or `/fr` routes like /:path* would
43     source: '/(.*)',
44     headers: [
45         {
46             key: 'x-hello',
47             value: 'world',
48         },
49     ],
50 },
51 ];
52 },
53 };
```

Cache-Control

You can set the `Cache-Control` header in your [Next.js API Routes](#) by using the `res.setHeader` method:

```
js pages/api/user.js

1 export default function handler(req, res) {
2     res.setHeader('Cache-Control', 's-maxage=86400');
3     res.status(200).json({ name: 'John Doe' });
4 }
```

You cannot set `Cache-Control` headers in `next.config.js` file as these will be overwritten in production to ensure that API Routes and static assets are cached effectively.

If you need to revalidate the cache of a page that has been [statically generated](#), you can do so by setting the `revalidate` prop in the page's `getStaticProps` function.

Options

X-DNS-Prefetch-Control

This header controls DNS prefetching, allowing browsers to proactively perform domain name resolution on external links, images, CSS, JavaScript, and more. This prefetching is performed in the background, so

the [DNS](#) is more likely to be resolved by the time the referenced items are needed. This reduces latency when the user clicks a link.

```
1  {
2    key: 'X-DNS-Prefetch-Control',
3    value: 'on'
4 }
```

Strict-Transport-Security

This header informs browsers it should only be accessed using HTTPS, instead of using HTTP. Using the configuration below, all present and future subdomains will use HTTPS for a `max-age` of 2 years. This blocks access to pages or subdomains that can only be served over HTTP.

If you're deploying to [Vercel](#), this header is not necessary as it's automatically added to all deployments unless you declare `headers` in your `next.config.js`.

```
1  {
2    key: 'Strict-Transport-Security',
3    value: 'max-age=63072000; includeSubDomains; preload'
4 }
```

X-XSS-Protection

This header stops pages from loading when they detect reflected cross-site scripting (XSS) attacks. Although this protection is not necessary when sites implement a strong [Content-Security-Policy](#) disabling the use of inline JavaScript (`'unsafe-inline'`), it can still provide protection for older web browsers that don't support CSP.

```
1  {
2    key: 'X-XSS-Protection',
3    value: '1; mode=block'
4 }
```

X-Frame-Options

This header indicates whether the site should be allowed to be displayed within an `iframe`. This can prevent against clickjacking attacks. This header has been superseded by CSP's `frame-ancestors` option, which has better support in modern browsers.

```
1  {
2    key: 'X-Frame-Options',
```

```
3   value: 'SAMEORIGIN'  
4 }
```

Permissions-Policy

This header allows you to control which features and APIs can be used in the browser. It was previously named [Feature-Policy](#). You can view the full list of permission options [here ↗](#).

```
1 {  
2   key: 'Permissions-Policy',  
3   value: 'camera=(), microphone=(), geolocation=(), browsing-topics=()'  
4 }
```

X-Content-Type-Options

This header prevents the browser from attempting to guess the type of content if the [Content-Type](#) header is not explicitly set. This can prevent XSS exploits for websites that allow users to upload and share files. For example, a user trying to download an image, but having it treated as a different [Content-Type](#) like an executable, which could be malicious. This header also applies to downloading browser extensions. The only valid value for this header is [nosniff](#).

```
1 {  
2   key: 'X-Content-Type-Options',  
3   value: 'nosniff'  
4 }
```

Referrer-Policy

This header controls how much information the browser includes when navigating from the current website (origin) to another. You can read about the different options [here ↗](#).

```
1 {  
2   key: 'Referrer-Policy',  
3   value: 'origin-when-cross-origin'  
4 }
```

Content-Security-Policy

This header helps prevent cross-site scripting (XSS), clickjacking and other code injection attacks. Content Security Policy (CSP) can specify allowed origins for content including scripts, stylesheets, images, fonts, objects, media (audio, video), iframes, and more.

You can read about the many different CSP options [here ↗](#).

You can add Content Security Policy directives using a template string.

```
1 // Before defining your Security Headers
2 // add Content Security Policy directives using a template string.
3
4 const ContentSecurityPolicy = `
5   default-src 'self';
6   script-src 'self';
7   child-src example.com;
8   style-src 'self' example.com;
9   font-src 'self';
10 `;
```

When a directive uses a keyword such as `self`, wrap it in single quotes `' '`.

In the header's value, replace the new line with a space.

```
1 {
2   key: 'Content-Security-Policy',
3   value: ContentSecurityPolicy.replace(/\s{2,}/g, ' ').trim()
4 }
```

Version History

Version

Changes

v13.3.0

missing added.

v10.2.0

has added.

v9.5.0

Headers added.

> Menu

Pages Router > ... > next.config.js Options > httpAgentOptions

httpAgentOptions

In Node.js versions prior to 18, Next.js automatically polyfills `fetch()` with [node-fetch](#) and enables [HTTP Keep-Alive](#) by default.

To disable HTTP Keep-Alive for all `fetch()` calls on the server-side, open `next.config.js` and add the `httpAgentOptions` config:

js next.config.js

```
1 module.exports = {
2   httpAgentOptions: {
3     keepAlive: false,
4   },
5 }
```


> Menu

Pages Router > ... > next.config.js Options > images

images

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure `next.config.js` with the following:

JS next.config.js

```
1 module.exports = {
2   images: {
3     loader: 'custom',
4     loaderFile: './my/image/loader.js',
5   },
6 }
```

This `loaderFile` must point to a file relative to the root of your Next.js application. The file must export a default function that returns a string, for example:

```
1 export default function myImageLoader({ src, width, quality }) {
2   return `https://example.com/${src}?w=${width}&q=${quality || 75}`;
3 }
```

Alternatively, you can use the `loader` prop to pass the function to each instance of `next/image`.

Example Loader Configuration

- [Akamai](#)
- [Cloudinary](#)
- [Cloudflare](#)
- [Contentful](#)
- [Fastly](#)

- [Gumlet](#)
- [ImageEngine](#)
- [Imgix](#)
- [Thumbor](#)

Akamai

```
1 // Docs: https://techdocs.akamai.com/ivm/reference/test-images-on-demand
2 export default function akamaiLoader({ src, width, quality }) {
3   return `https://example.com/${src}?imwidth=${width}`;
4 }
```

Cloudinary

```
1 // Demo: https://res.cloudinary.com/demo/image/upload/w_300,c_limit,q_auto/turtles.jpg
2 export default function cloudinaryLoader({ src, width, quality }) {
3   const params = ['f_auto', 'c_limit', `w_${width}`, `q_${quality || 'auto'}`];
4   return `https://example.com/${params.join(',')}${src}`;
5 }
```

Cloudflare

```
1 // Docs: https://developers.cloudflare.com/images/url-format
2 export default function cloudflareLoader({ src, width, quality }) {
3   const params = [`width=${width}`, `quality=${quality || 75}`, 'format=auto'];
4   return `https://example.com/cdn-cgi/image/${params.join(',')}/${src}`;
5 }
```

Contentful

```
1 // Docs: https://www.contentful.com/developers/docs/references/images-api/
2 export default function contentfulLoader({ src, quality, width }) {
3   const url = new URL(`https://example.com${src}`);
4   urlSearchParams.set('fm', 'webp');
5   urlSearchParams.set('w', width.toString());
6   urlSearchParams.set('q', quality.toString() || '75');
7   return url.href;
8 }
```

Fastly

```
1 // Docs: https://developer.fastly.com/reference/io/
2 export default function fastlyLoader({ src, width, quality }) {
3   const url = new URL(`https://example.com${src}`);
4   urlSearchParams.set('auto', 'webp');
5   urlSearchParams.set('width', width.toString());
6   urlSearchParams.set('quality', quality.toString() || '75');
7   return url.href;
8 }
```

Gumlet

```
1 // Docs: https://docs.gumlet.com/reference/image-transform-size
2 export default function gumletLoader({ src, width, quality }) {
3   const url = new URL(`https://example.com${src}`);
4   urlSearchParams.set('format', 'auto');
5   urlSearchParams.set('w', width.toString());
6   urlSearchParams.set('q', quality.toString() || '75');
7   return url.href;
8 }
```

ImageEngine

```
1 // Docs: https://support.imageengine.io/hc/en-us/articles/360058880672-Directives
2 export default function imageengineLoader({ src, width, quality }) {
3   const compression = 100 - (quality || 50)
4   const params = [ `w_${width}`, `cmpr_${compression}` ]
5   return `https://example.com${src}?imgeng=/${params.join('/')}`
6 }
```

Imgix

```
1 // Demo: https://static.imgix.net/daisy.png?format=auto&fit=max&w=300
2 export default function imgixLoader({ src, width, quality }) {
3   const url = new URL(`https://example.com${src}`);
4   const params = url.searchParams;
5   params.set('auto', params.getAll('auto').join(',') || 'format');
6   params.set('fit', params.get('fit') || 'max');
7   params.set('w', params.get('w') || width.toString());
8   params.set('q', quality.toString() || '50');
```

```
9   return url.href;  
10 }
```

Thumbor

```
1 // Docs: https://thumbor.readthedocs.io/en/latest/  
2 export default function thumborLoader({ src, width, quality }) {  
3   const params = [`width${width}x0`, `filters:quality(${quality || 75})`];  
4   return `https://example.com${params.join('/')}${src}`;  
5 }
```

> Menu

Pages Router > ... > next.config.js Options > onDemandEntries

onDemandEntries

Next.js exposes some options that give you some control over how the server will dispose or keep in memory built pages in development.

To change the defaults, open `next.config.js` and add the `onDemandEntries` config:

JS next.config.js



```
1 module.exports = {
2   onDemandEntries: {
3     // period (in ms) where the server will keep pages in the buffer
4     maxInactiveAge: 25 * 1000,
5     // number of pages that should be kept simultaneously without being disposed
6     pagesBufferLength: 2,
7   },
8 };
```


> Menu

Pages Router > ... > next.config.js Options > output

output

During a build, Next.js will automatically trace each page and its dependencies to determine all of the files that are needed for deploying a production version of your application.

This feature helps reduce the size of deployments drastically. Previously, when deploying with Docker you would need to have all files from your package's `dependencies` installed to run `next start`. Starting with Next.js 12, you can leverage Output File Tracing in the `.next/` directory to only include the necessary files.

Furthermore, this removes the need for the deprecated `serverless` target which can cause various issues and also creates unnecessary duplication.

How it Works

During `next build`, Next.js will use [@vercel/nft](#) to statically analyze `import`, `require`, and `fs` usage to determine all files that a page might load.

Next.js' production server is also traced for its needed files and output at `.next/next-server.js.nft.json` which can be leveraged in production.

To leverage the `.nft.json` files emitted to the `.next` output directory, you can read the list of files in each trace that are relative to the `.nft.json` file and then copy them to your deployment location.

Automatically Copying Traced Files

Next.js can automatically create a `standalone` folder that copies only the necessary files for a production deployment including select files in `node_modules`.

To leverage this automatic copying you can enable it in your `next.config.js`:



```
1 module.exports = {  
2   output: 'standalone',  
3 };
```

This will create a folder at `.next/standalone` which can then be deployed on its own without installing `node_modules`.

Additionally, a minimal `server.js` file is also output which can be used instead of `next start`. This minimal server does not copy the `public` or `.next/static` folders by default as these should ideally be handled by a CDN instead, although these folders can be copied to the `standalone/public` and `standalone/.next/static` folders manually, after which `server.js` file will serve these automatically.

Note: `next.config.js` is read during `next build` and serialized into the `server.js` output file. If the legacy `serverRuntimeConfig` or `publicRuntimeConfig` options are being used, the values will be specific to values at build time.

Note: If your project uses [Image Optimization](#) with the default `loader`, you must install `sharp` as a dependency:

>_ Terminal



```
npm i sharp
```

>_ Terminal



```
yarn add sharp
```

>_ Terminal



```
pnpm add sharp
```

Caveats

- While tracing in monorepo setups, the project directory is used for tracing by default. For `next build` `packages/web-app`, `packages/web-app` would be the tracing root and any files outside of that folder

will not be included. To include files outside of this folder you can set `experimental.outputFileTracingRoot` in your `next.config.js`.

JS packages/web-app/next.config.js

```
1 module.exports = {
2   experimental: {
3     // this includes files from the monorepo base two directories up
4     outputFileTracingRoot: path.join(__dirname, '../..'),
5   },
6 }
```

- There are some cases in which Next.js might fail to include required files, or might incorrectly include unused files. In those cases, you can leverage `experimental.outputFileTracingExcludes` and `experimental.outputFileTracingIncludes` respectively in `next.config.js`. Each config accepts an object with [minimatch globs](#) ↗ for the key to match specific pages and a value of an array with globs relative to the project's root to either include or exclude in the trace.

JS next.config.js

```
1 module.exports = {
2   experimental: {
3     outputFileTracingExcludes: {
4       '/api/hello': ['./un-necessary-folder/**/*'],
5     },
6     outputFileTracingIncludes: {
7       '/api/another': ['./necessary-folder/**/*'],
8     },
9   },
10 }
```

- Currently, Next.js does not do anything with the emitted `.nft.json` files. The files must be read by your deployment platform, for example [Vercel](#) ↗, to create a minimal deployment. In a future release, a new command is planned to utilize these `.nft.json` files.

Experimental `turbotrace`

Tracing dependencies can be slow because it requires very complex computations and analysis. We created `turbotrace` in Rust as a faster and smarter alternative to the JavaScript implementation.

To enable it, you can add the following configuration to your `next.config.js`:



```
1 module.exports = {
2   experimental: {
3     turbotrace: {
4       // control the log level of the turbotrace, default is `error`
5       logLevel?: [
6         'bug'
7         | 'fatal'
8         | 'error'
9         | 'warning'
10        | 'hint'
11        | 'note'
12        | 'suggestions'
13        | 'info',
14       // control if the log of turbotrace should contain the details of the analysis, def
15       logDetail?: boolean
16       // show all log messages without limit
17       // turbotrace only show 1 log message for each categories by default
18       logAll?: boolean
19       // control the context directory of the turbotrace
20       // files outside of the context directory will not be traced
21       // set the `experimental.outputFileTracingRoot` has the same effect
22       // if the `experimental.outputFileTracingRoot` and this option are both set, the `e
23       contextDirectory?: string
24       // if there is `process.cwd()` expression in your code, you can set this option to
25       // for example the require(process.cwd() + '/package.json') will be traced as requi
26       processCwd?: string
27       // control the maximum memory usage of the `turbotrace`, in `MB`, default is `6000`-
28       memoryLimit?: number
29     },
30   },
31 }
```

> Menu

Pages Router > ... > next.config.js Options > pageExtensions

pageExtensions

You can extend the default Page extensions (`.tsx` , `.ts` , `.jsx` , `.js`) used by Next.js. Inside `next.config.js` , add the `pageExtensions` config:

JS next.config.js

```
1 module.exports = {  
2   pageExtensions: ['mdx', 'md', 'jsx', 'js', 'tsx', 'ts'],  
3 };
```



Changing these values affects *all* Next.js pages, including the following:

- `middleware.js`
- `pages/_document.js`
- `pages/_app.js`
- `pages/api/`

For example, if you reconfigure `.ts` page extensions to `.page.ts` , you would need to rename pages like `_app.page.ts` .

Including non-page files in the `pages` directory

You can colocate test files or other files used by components in the `pages` directory. Inside `next.config.js` , add the `pageExtensions` config:

JS next.config.js

```
1 module.exports = {  
2   pageExtensions: ['page.tsx', 'page.ts', 'page.jsx', 'page.js'],
```



3 };

Then, rename your pages to have a file extension that includes `.page` (e.g. rename `MyPage.tsx` to `MyPage.page.tsx`). Ensure you rename *all* Next.js pages, including the files mentioned above.

> Menu

Pages Router > ... > next.config.js Options > poweredByHeader

poweredByHeader

By default Next.js will add the `x-powered-by` header. To opt-out of it, open `next.config.js` and disable the `poweredByHeader` config:

JS next.config.js

```
1 module.exports = {  
2   poweredByHeader: false,  
3 };
```


> Menu

Pages Router > ... > next.config.js Options > productionBrowserSourceMaps

productionBrowserSourceMaps

Source Maps are enabled by default during development. During production builds, they are disabled to prevent you leaking your source on the client, unless you specifically opt-in with the configuration flag.

Next.js provides a configuration flag you can use to enable browser source map generation during the production build:

js next.config.js

```
1 module.exports = {  
2   productionBrowserSourceMaps: true,  
3 };
```

When the `productionBrowserSourceMaps` option is enabled, the source maps will be output in the same directory as the JavaScript files. Next.js will automatically serve these files when requested.

- Adding source maps can increase `next build` time
- Increases memory usage during `next build`

> Menu

Pages Router > ... > next.config.js Options > reactStrictMode

reactStrictMode

Suggested: We strongly suggest you enable Strict Mode in your Next.js application to better prepare your application for the future of React.

React's [Strict Mode](#) is a development mode only feature for highlighting potential problems in an application. It helps to identify unsafe lifecycles, legacy API usage, and a number of other features.

The Next.js runtime is Strict Mode-compliant. To opt-in to Strict Mode, configure the following option in your `next.config.js`:

JS next.config.js



```
1 module.exports = {  
2   reactStrictMode: true,  
3 };
```

If you or your team are not ready to use Strict Mode in your entire application, that's OK! You can incrementally migrate on a page-by-page basis using `<React.StrictMode>`.

> Menu

Pages Router > ... > next.config.js Options > redirects

redirects

Redirects allow you to redirect an incoming request path to a different destination path.

To use redirects you can use the `redirects` key in `next.config.js`:

js next.config.js

```
1 module.exports = {
2   async redirects() {
3     return [
4       {
5         source: '/about',
6         destination: '/',
7         permanent: true,
8       },
9     ];
10   },
11};
```



`redirects` is an `async` function that expects an array to be returned holding objects with `source`, `destination`, and `permanent` properties:

- `source` is the incoming request path pattern.
- `destination` is the path you want to route to.
- `permanent` `true` or `false` - if `true` will use the 308 status code which instructs clients/search engines to cache the redirect forever, if `false` will use the 307 status code which is temporary and is not cached.

Why does Next.js use 307 and 308? Traditionally a 302 was used for a temporary redirect, and a 301 for a permanent redirect, but many browsers changed the request method of the redirect to `GET`, regardless of the original method. For example, if the browser made a request to `POST /v1/users` which returned status code `302` with location `/v2/users`, the subsequent request might be `GET /v2/users` instead of the expected `POST /v2/users`. Next.js uses the 307 temporary redirect, and 308 permanent redirect status codes to explicitly preserve the request method used.

- `basePath`: `false` or `undefined` - if false the `basePath` won't be included when matching, can be used for external redirects only.
- `locale`: `false` or `undefined` - whether the locale should not be included when matching.
- `has` is an array of `has objects` with the `type`, `key` and `value` properties.
- `missing` is an array of `missing objects` with the `type`, `key` and `value` properties.

Redirects are checked before the filesystem which includes pages and `/public` files.

Redirects are not applied to client-side routing (`Link`, `router.push`), unless `Middleware` is present and matches the path.

When a redirect is applied, any query values provided in the request will be passed through to the redirect destination. For example, see the following redirect configuration:

```

1  {
2    source: '/old-blog/:path*',
3    destination: '/blog/:path*',
4    permanent: false
5  }

```

When `/old-blog/post-1?hello=world` is requested, the client will be redirected to `/blog/post-1?hello=world`.

Path Matching

Path matches are allowed, for example `/old-blog/:slug` will match `/old-blog/hello-world` (no nested paths):

`JS` `next.config.js`

```

1  module.exports = {
2    async redirects() {
3      return [
4        {
5          source: '/old-blog/:slug',
6          destination: '/news/:slug', // Matched parameters can be used in the destination
7          permanent: true,
8        },
9      ];
10    },
11  };

```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:

JS next.config.js

```
1 module.exports = {
2   async redirects() {
3     return [
4       {
5         source: '/blog/:slug*',
6         destination: '/news/:slug*', // Matched parameters can be used in the destination
7         permanent: true,
8       },
9     ];
10   },
11 };
```

Regex Path Matching

To match a regex path you can wrap the regex in parentheses after a parameter, for example

`/post/:slug(\d{1,})` will match `/post/123` but not `/post/abc`:

JS next.config.js

```
1 module.exports = {
2   async redirects() {
3     return [
4       {
5         source: '/post/:slug(\d{1,})',
6         destination: '/news/:slug', // Matched parameters can be used in the destination
7         permanent: false,
8       },
9     ];
10   },
11 };
```

The following characters `(`, `)`, `{`, `}`, `:`, `*`, `+`, `?` are used for regex path matching, so when used in the `source` as non-special values they must be escaped by adding `\` before them:

JS next.config.js

```
1 module.exports = {
2   async redirects() {
3     return [
4       {
```

```
5 // this will match `/english(default)/something` being requested
6 source: '/english\\(default\\)\\/slug',
7 destination: '/en-us/:slug',
8 permanent: false,
9 },
10 ],
11 },
12 };
```

Header, Cookie, and Query Matching

To only match a redirect when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the redirect to be applied.

`has` and `missing` items can have the following fields:

- `type: String` - must be either `header`, `cookie`, `host`, or `query`.
- `key: String` - the key from the selected type to match against.
- `value: String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

js next.config.js

```
1 module.exports = {
2   async redirects() {
3     return [
4       // if the header `x-redirect-me` is present,
5       // this redirect will be applied
6       {
7         source: '/:path((?!another-page$).*)',
8         has: [
9           {
10             type: 'header',
11             key: 'x-redirect-me',
12           },
13         ],
14         permanent: false,
15         destination: '/another-page',
16       },
17       // if the header `x-dont-redirect` is present,
18       // this redirect will NOT be applied
19       {
20         source: '/:path((?!another-page$).*)',
```

```
21 missing: [
22   {
23     type: 'header',
24     key: 'x-do-not-redirect',
25   },
26 ],
27 permanent: false,
28 destination: '/another-page',
29 },
30 // if the source, query, and cookie are matched,
31 // this redirect will be applied
32 {
33   source: '/specific/:path*',
34   has: [
35     {
36       type: 'query',
37       key: 'page',
38       // the page value will not be available in the
39       // destination since value is provided and doesn't
40       // use a named capture group e.g. (?<page>home)
41       value: 'home',
42     },
43     {
44       type: 'cookie',
45       key: 'authorized',
46       value: 'true',
47     },
48   ],
49   permanent: false,
50   destination: '/another/:path*',
51 },
52 // if the header `x-authorized` is present and
53 // contains a matching value, this redirect will be applied
54 {
55   source: '/',
56   has: [
57     {
58       type: 'header',
59       key: 'x-authorized',
60       value: '(?<authorized>yes|true)',
61     },
62   ],
63   permanent: false,
64   destination: '/home?authorized=:authorized',
65 },
66 // if the host is `example.com`,
67 // this redirect will be applied
68 {
69   source: '/:path((?!another-page$).*)',
70   has: [
71     {
72       type: 'host',
73       value: 'example.com',
74     },
75   ],
76   permanent: false,
```

```
77         destination: '/another-page',
78     },
79   ];
80 },
81 };
```

Redirects with basePath support

When leveraging `basePath` support with redirects each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the redirect:

js next.config.js

```
1 module.exports = {
2   basePath: '/docs',
3
4   async redirects() {
5     return [
6       {
7         source: '/with-basePath', // automatically becomes /docs/with-basePath
8         destination: '/another', // automatically becomes /docs/another
9         permanent: false,
10      },
11      {
12        // does not add /docs since basePath: false is set
13        source: '/without-basePath',
14        destination: 'https://example.com',
15        basePath: false,
16        permanent: false,
17      },
18    ];
19  },
20};
```

Redirects with i18n support

When leveraging `i18n` support with redirects each `source` and `destination` is automatically prefixed to handle the configured `locales` unless you add `locale: false` to the redirect. If `locale: false` is used you must prefix the `source` and `destination` with a locale for it to be matched correctly.

js next.config.js

```
1 module.exports = {
2   i18n: {
3     locales: ['en', 'fr', 'de'],
4     defaultLocale: 'en',
5   },
6 }
```

```

7   async redirects() {
8     return [
9       {
10         source: '/with-locale', // automatically handles all locales
11         destination: '/another', // automatically passes the locale on
12         permanent: false,
13       },
14       {
15         // does not handle locales automatically since locale: false is set
16         source: '/nl/with-locale-manual',
17         destination: '/nl/another',
18         locale: false,
19         permanent: false,
20       },
21       {
22         // this matches '/' since `en` is the defaultLocale
23         source: '/en',
24         destination: '/en/another',
25         locale: false,
26         permanent: false,
27       },
28       // it's possible to match all locales even when locale: false is set
29       {
30         source: '/:locale/page',
31         destination: '/en/newpage',
32         permanent: false,
33         locale: false,
34       }
35       {
36         // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
37         // `/` or `/fr` routes like `/:path*` would
38         source: '/(.*)',
39         destination: '/another',
40         permanent: false,
41       },
42     ]
43   },
44 }

```

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. To ensure IE11 compatibility, a `Refresh` header is automatically added for the 308 status code.

Other Redirects

- Inside `API Routes`, you can use `res.redirect()`.
- Inside `getStaticProps` and `getServerSideProps`, you can redirect specific pages at request-time.

Version History

Version	Changes
v13.3.0	<code>missing</code> added.
v10.2.0	<code>has</code> added.
v9.5.0	<code>redirects</code> added.

> Menu

Pages Router > ... > next.config.js Options > rewrites

rewrites

Rewrites allow you to map an incoming request path to a different destination path.

Rewrites act as a URL proxy and mask the destination path, making it appear the user hasn't changed their location on the site. In contrast, [redirects](#) will reroute to a new page and show the URL changes.

To use rewrites you can use the `rewrites` key in `next.config.js`:

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/about',
6         destination: '/',
7       },
8     ];
9   },
10};
```



Rewrites are applied to client-side routing, a `<Link href="/about">` will have the rewrite applied in the above example.

`rewrites` is an `async` function that expects to return either an array or an object of arrays (see below) holding objects with `source` and `destination` properties:

- `source : String` - is the incoming request path pattern.
- `destination : String` is the path you want to route to.
- `basePath: false` or `undefined` - if false the basePath won't be included when matching, can be used for external rewrites only.
- `locale: false` or `undefined` - whether the locale should not be included when matching.
- `has` is an array of `has objects` with the `type`, `key` and `value` properties.

- `missing` is an array of `missing objects` with the `type`, `key` and `value` properties.

When the `rewrites` function returns an array, rewrites are applied after checking the filesystem (pages and `/public` files) and before dynamic routes. When the `rewrites` function returns an object of arrays with a specific shape, this behavior can be changed and more finely controlled, as of `v10.1` of Next.js:

`JS` next.config.js

```

1 module.exports = {
2   async rewrites() {
3     return {
4       beforeFiles: [
5         // These rewrites are checked after headers/redirects
6         // and before all files including _next/public files which
7         // allows overriding page files
8         {
9           source: '/some-page',
10          destination: '/somewhere-else',
11          has: [{ type: 'query', key: 'overrideMe' }],
12        },
13      ],
14      afterFiles: [
15        // These rewrites are checked after pages/public files
16        // are checked but before dynamic routes
17        {
18          source: '/non-existent',
19          destination: '/somewhere-else',
20        },
21      ],
22      fallback: [
23        // These rewrites are checked after both pages/public files
24        // and dynamic routes are checked
25        {
26          source: '/:path*',
27          destination: `https://my-old-site.com/:path*`,
28        },
29      ],
30    };
31  },
32};
```

Note: rewrites in `beforeFiles` do not check the filesystem/dynamic routes immediately after matching a source, they continue until all `beforeFiles` have been checked.

The order Next.js routes are checked is:

1. `headers` are checked/applied
2. `redirects` are checked/applied

3. `beforeFiles` rewrites are checked/applied
4. static files from the [public directory](#), `_next/static` files, and non-dynamic pages are checked/served
5. `afterFiles` rewrites are checked/applied, if one of these rewrites is matched we check dynamic routes/static files after each match
6. `fallback` rewrites are checked/applied, these are applied before rendering the 404 page and after dynamic routes/all static assets have been checked. If you use `fallback: true/'blocking'` in `getStaticPaths`, the fallback `rewrites` defined in your `next.config.js` will *not* be run.

Rewrite parameters

When using parameters in a rewrite the parameters will be passed in the query by default when none of the parameters are used in the `destination`.

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/old-about/:path*',
6         destination: '/about', // The :path parameter isn't used here so will be automatically passed in the query
7       },
8     ];
9   },
10};
```

If a parameter is used in the destination none of the parameters will be automatically passed in the query.

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/docs/:path*',
6         destination: '/:path*', // The :path parameter is used here so will not be automatically passed in the query
7       },
8     ];
9   },
10};
```

You can still pass the parameters manually in the query if one is already used in the destination by specifying the query in the `destination`.

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/:first/:second',
6         destination: '/:first?second=:second',
7         // Since the :first parameter is used in the destination the :second parameter
8         // will not automatically be added in the query although we can manually add it
9         // as shown above
10      },
11    ];
12  },
13};
```

Note: Static pages from [Automatic Static Optimization](#) or [prerendering](#) params from rewrites will be parsed on the client after hydration and provided in the query.

Path Matching

Path matches are allowed, for example `/blog/:slug` will match `/blog/hello-world` (no nested paths):

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/blog/:slug',
6         destination: '/news/:slug', // Matched parameters can be used in the destination
7       },
8     ];
9   },
10};
```

Wildcard Path Matching

To match a wildcard path you can use `*` after a parameter, for example `/blog/:slug*` will match `/blog/a/b/c/d/hello-world`:

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/blog/:slug*',
6         destination: '/news/:slug*', // Matched parameters can be used in the destination
7       },
8     ];
9   },
10};
```

Regex Path Matching

To match a regex path you can wrap the regex in parenthesis after a parameter, for example

/blog/:slug(\d{1,}) will match /blog/123 but not /blog/abc:

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/old-blog/:post(\d{1,})',
6         destination: '/blog/:post', // Matched parameters can be used in the destination
7       },
8     ];
9   },
10};
```

The following characters (,), {, }, :, *, +, ? are used for regex path matching, so when used in the `source` as non-special values they must be escaped by adding \ before them:

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         // this will match `/english(default)/something` being requested
6         source: '/english\\(default\\)/:slug',
7         destination: '/en-us/:slug',
8       },
9     ];
10   },
11};
```

Header, Cookie, and Query Matching

To only match a rewrite when header, cookie, or query values also match the `has` field or don't match the `missing` field can be used. Both the `source` and all `has` items must match and all `missing` items must not match for the rewrite to be applied.

`has` and `missing` items can have the following fields:

- `type : String` - must be either `header`, `cookie`, `host`, or `query`.
- `key : String` - the key from the selected type to match against.
- `value : String` or `undefined` - the value to check for, if undefined any value will match. A regex like string can be used to capture a specific part of the value, e.g. if the value `first-(?<paramName>.*)` is used for `first-second` then `second` will be usable in the destination with `:paramName`.

js next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       // if the header `x-rewrite-me` is present,
5       // this rewrite will be applied
6       {
7         source: '/:path*',
8         has: [
9           {
10             type: 'header',
11             key: 'x-rewrite-me',
12           },
13           ],
14         destination: '/another-page',
15       },
16       // if the header `x-rewrite-me` is not present,
17       // this rewrite will be applied
18       {
19         source: '/:path*',
20         missing: [
21           {
22             type: 'header',
23             key: 'x-rewrite-me',
24           },
25           ],
26         destination: '/another-page',
27       },
28       // if the source, query, and cookie are matched,
29       // this rewrite will be applied
30       {
31         source: '/specific/:path*',
```

```

32     has: [
33         {
34             type: 'query',
35             key: 'page',
36             // the page value will not be available in the
37             // destination since value is provided and doesn't
38             // use a named capture group e.g. (?<page>home)
39             value: 'home',
40         },
41         {
42             type: 'cookie',
43             key: 'authorized',
44             value: 'true',
45         },
46     ],
47     destination: '/:path*/home',
48 },
49 // if the header `x-authorized` is present and
50 // contains a matching value, this rewrite will be applied
51 {
52     source: '/:path*',
53     has: [
54         {
55             type: 'header',
56             key: 'x-authorized',
57             value: '(?<authorized>yes|true)',
58         },
59     ],
60     destination: '/home?authorized=:authorized',
61 },
62 // if the host is `example.com`,
63 // this rewrite will be applied
64 {
65     source: '/:path*',
66     has: [
67         {
68             type: 'host',
69             value: 'example.com',
70         },
71     ],
72     destination: '/another-page',
73 },
74 ];
75 },
76 };

```

Rewriting to an external URL

► Examples

Rewrites allow you to rewrite to an external url. This is especially useful for incrementally adopting Next.js. The following is an example rewrite for redirecting the `/blog` route of your main app to an external site.

JS next.config.js

```
1 module.exports = {
2   async rewrites() {
3     return [
4       {
5         source: '/blog',
6         destination: 'https://example.com/blog',
7       },
8       {
9         source: '/blog/:slug',
10        destination: 'https://example.com/blog/:slug', // Matched parameters can be used
11      },
12    ];
13  },
14};
```

If you're using `trailingSlash: true`, you also need to insert a trailing slash in the `source` parameter. If the destination server is also expecting a trailing slash it should be included in the `destination` parameter as well.

JS next.config.js

```
1 module.exports = {
2   trailingSlash: true,
3   async rewrites() {
4     return [
5       {
6         source: '/blog/',
7         destination: 'https://example.com/blog/',
8       },
9       {
10         source: '/blog/:path*/',
11         destination: 'https://example.com/blog/:path*/',
12       },
13     ];
14  },
15};
```

Incremental adoption of Next.js

You can also have Next.js fall back to proxying to an existing website after checking all Next.js routes.

This way you don't have to change the rewrites configuration when migrating more pages to Next.js



```

1 module.exports = {
2   async rewrites() {
3     return {
4       fallback: [
5         {
6           source: '/:path*',
7           destination: `https://custom-routes-proxying-endpoint.vercel.app/:path*`,
8         },
9       ],
10      };
11    },
12  };

```

Rewrites with basePath support

When leveraging `basePath` support with rewrites each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the rewrite:



```

1 module.exports = {
2   basePath: '/docs',
3
4   async rewrites() {
5     return [
6       {
7         source: '/with-basePath', // automatically becomes /docs/with-basePath
8         destination: '/another', // automatically becomes /docs/another
9       },
10      {
11        // does not add /docs to /without-basePath since basePath: false is set
12        // Note: this can not be used for internal rewrites e.g. `destination: '/another'
13        source: '/without-basePath',
14        destination: 'https://example.com',
15        basePath: false,
16      },
17    ];
18  },
19};

```

Rewrites with i18n support

When leveraging `i18n` support with rewrites each `source` and `destination` is automatically prefixed to handle the configured `locales` unless you add `locale: false` to the rewrite. If `locale: false` is used you must prefix the `source` and `destination` with a locale for it to be matched correctly.

```
1 module.exports = {
2   i18n: {
3     locales: ['en', 'fr', 'de'],
4     defaultLocale: 'en',
5   },
6
7   async rewrites() {
8     return [
9       {
10         source: '/with-locale', // automatically handles all locales
11         destination: '/another', // automatically passes the locale on
12       },
13       {
14         // does not handle locales automatically since locale: false is set
15         source: '/nl/with-locale-manual',
16         destination: '/nl/another',
17         locale: false,
18       },
19       {
20         // this matches '/' since `en` is the defaultLocale
21         source: '/en',
22         destination: '/en/another',
23         locale: false,
24       },
25       {
26         // it's possible to match all locales even when locale: false is set
27         source: '/:locale/api-alias/:path*',
28         destination: '/api/:path*',
29         locale: false,
30       },
31       {
32         // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
33         // `/` or `/fr` routes like `/:path*` would
34         source: '/(.*)',
35         destination: '/another',
36       },
37     ];
38   },
39 };
```

Version History

Version

Changes

v13.3.0

missing added.

v10.2.0

has added.

Version**Changes**

v9.5.0

Headers added.

> Menu

Pages Router > ... > next.config.js Options > Runtime Config

Runtime Config

Note: This feature is considered legacy and does not work with [Automatic Static Optimization](#), [Output File Tracing](#), or [React Server Components](#). Please use [environment variables](#) instead to avoid initialization overhead.

To add runtime configuration to your app, open `next.config.js` and add the `publicRuntimeConfig` and `serverRuntimeConfig` configs:

JS next.config.js

```
1 module.exports = {
2   serverRuntimeConfig: {
3     // Will only be available on the server side
4     mySecret: 'secret',
5     secondSecret: process.env.SECOND_SECRET, // Pass through env variables
6   },
7   publicRuntimeConfig: {
8     // Will be available on both server and client
9     staticFolder: '/static',
10   },
11 };
```

Place any server-only runtime config under `serverRuntimeConfig`.

Anything accessible to both client and server-side code should be under `publicRuntimeConfig`.

A page that relies on `publicRuntimeConfig` **must** use `getInitialProps` or `getServerSideProps` or your application must have a [Custom App](#) with `getInitialProps` to opt-out of [Automatic Static Optimization](#). Runtime configuration won't be available to any page (or component in a page) without being server-side rendered.

To get access to the runtime configs in your app use `next/config`, like so:

```
1 import getConfig from 'next/config';
2 import Image from 'next/image';
3
```

```
4 // Only holds serverRuntimeConfig and publicRuntimeConfig
5 const { serverRuntimeConfig, publicRuntimeConfig } = getConfig();
6 // Will only be available on the server-side
7 console.log(serverRuntimeConfig.mySecret);
8 // Will be available on both server-side and client-side
9 console.log(publicRuntimeConfig.staticFolder);
10
11 function MyImage() {
12     return (
13         <div>
14             <Image
15                 src={`${publicRuntimeConfig.staticFolder}/logo.png`}
16                 alt="logo"
17                 layout="fill"
18             />
19         </div>
20     );
21 }
22
23 export default MyImage;
```

> Menu

Pages Router > ... > next.config.js Options > trailingSlash

trailingSlash

By default Next.js will redirect urls with trailing slashes to their counterpart without a trailing slash. For example `/about/` will redirect to `/about`. You can configure this behavior to act the opposite way, where urls without trailing slashes are redirected to their counterparts with trailing slashes.

Open `next.config.js` and add the `trailingSlash` config:

JS next.config.js

```
1 module.exports = {  
2   trailingSlash: true,  
3 };
```

With this option set, urls like `/about` will redirect to `/about/`.

Version History

Version	Changes
v9.5.0	<code>trailingSlash</code> added.

> Menu

Pages Router > ... > next.config.js Options > transpilePackages

transpilePackages

Next.js can automatically transpile and bundle dependencies from local packages (like monorepos) or from external dependencies (node_modules). This replaces the `next-transpile-modules` package.

JS next.config.js



```
1  /** @type {import('next').NextConfig} */
2  const nextConfig = {
3    transpilePackages: ['@acme/ui', 'lodash-es'],
4  };
5
6 module.exports = nextConfig;
```

Version History

Version**Changes**

v13.0.0

transpilePackages added.

> Menu

Pages Router > ... > next.config.js Options > turbo

turbo (experimental)

Warning: These features are experimental and will only work with `next --turbo`.

webpack loaders

Currently, Turbopack supports a subset of webpack's loader API, allowing you to use some webpack loaders to transform code in Turbopack.

To configure loaders, add the names of the loaders you've installed and any options in `next.config.js`, mapping file extensions to a list of loaders:

JS next.config.js



```
1 module.exports = {
2   experimental: {
3     turbo: {
4       loaders: {
5         // Option format
6         '.md': [
7           {
8             loader: '@mdx-js/loader',
9             options: {
10               format: 'md',
11             },
12           },
13         ],
14         // Option-less format
15         '.mdx': ['@mdx-js/loader'],
16       },
17     },
18   },
19 };
```

Then, given the above configuration, you can use transformed code from your app:

```
1 import MyDoc from './my-doc.mdx';
2
3 export default function Home() {
4   return <MyDoc />;
5 }
```

Resolve Alias

Through `next.config.js`, Turbopack can be configured to modify module resolution through aliases, similar to webpack's `resolve.alias` [configuration](#).

To configure resolve aliases, map imported patterns to their new destination in `next.config.js`:

js next.config.js

```
1 module.exports = {
2   experimental: {
3     turbo: {
4       resolveAlias: {
5         underscore: 'lodash',
6         mocha: { browser: 'mocha/browser-entry.js' },
7       },
8     },
9   },
10};
```

This aliases imports of the `underscore` package to the `lodash` package. In other words, `import underscore from 'underscore'` will load the `lodash` module instead of `underscore`.

Turbopack also supports conditional aliasing through this field, similar to Node.js's `conditional exports` [↗](#). At the moment only the `browser` condition is supported. In the case above, imports of the `mocha` module will be aliased to `mocha/browser-entry.js` when Turbopack targets browser environments.

For more information and guidance for how to migrate your app to Turbopack from webpack, see [Turbopack's documentation on webpack compatibility](#) [↗](#).

> Menu

Pages Router > ... > next.config.js Options > typescript

typescript

Next.js fails your **production build** (`next build`) when TypeScript errors are present in your project.

If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.

If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.

Open `next.config.js` and enable the `ignoreBuildErrors` option in the `typescript` config:

JS next.config.js



```
1 module.exports = {
2   typescript: {
3     // !!! WARN !!!
4     // Dangerously allow production builds to successfully complete even if
5     // your project has type errors.
6     // !!! WARN !!!
7     ignoreBuildErrors: true,
8   },
9 }
```


> Menu

Pages Router > ... > next.config.js Options > urlImports

urlImports

URL imports are an experimental feature that allows you to import modules directly from external servers (instead of from the local disk).

Warning: This feature is experimental. Only use domains that you trust to download and execute on your machine. Please exercise discretion, and caution until the feature is flagged as stable.

To opt-in, add the allowed URL prefixes inside `next.config.js`:

JS next.config.js

```
1 module.exports = {
2   experimental: {
3     urlImports: ['https://example.com/assets/', 'https://cdn.skypack.dev'],
4   },
5 };
```

Then, you can import modules directly from URLs:

```
import { a, b, c } from 'https://example.com/assets/some/module.js';
```

URL Imports can be used everywhere normal package imports can be used.

Security Model

This feature is being designed with **security as the top priority**. To start, we added an experimental flag forcing you to explicitly allow the domains you accept URL imports from. We're working to take this further by limiting URL imports to execute in the browser sandbox using the [Edge Runtime](#).

Lockfile

When using URL imports, Next.js will create a `next.lock` directory containing a lockfile and fetched assets. This directory **must be committed to Git**, not ignored by `.gitignore`.

- When running `next dev`, Next.js will download and add all newly discovered URL Imports to your lockfile
- When running `next build`, Next.js will use only the lockfile to build the application for production

Typically, no network requests are needed and any outdated lockfile will cause the build to fail. One exception is resources that respond with `Cache-Control: no-cache`. These resources will have a `no-cache` entry in the lockfile and will always be fetched from the network on each build.

Examples

Skypack

```
1 import confetti from 'https://cdn.skypack.dev/canvas-confetti';
2 import { useEffect } from 'react';
3
4 export default () => {
5   useEffect(() => {
6     confetti();
7   });
8   return <p>Hello</p>;
9 }
```

Static Image Imports

```
1 import Image from 'next/image';
2 import logo from 'https://example.com/assets/logo.png';
3
4 export default () => (
5   <div>
6     <Image src={logo} placeholder="blur" />
7   </div>
8 );
```

URLs in CSS

```
1 .className {  
2   background: url('https://example.com/assets/hero.jpg');  
3 }
```

Asset Imports

```
1 const logo = new URL('https://example.com/assets/file.txt', import.meta.url);  
2  
3 console.log(logo.pathname);  
4  
5 // prints "/_next/static/media/file.a9727b5d.txt"
```

> Menu

Pages Router > ... > next.config.js Options > webVitalsAttribution

webVitalsAttribution

When debugging issues related to Web Vitals, it is often helpful if we can pinpoint the source of the problem. For example, in the case of Cumulative Layout Shift (CLS), we might want to know the first element that shifted when the single largest layout shift occurred. Or, in the case of Largest Contentful Paint (LCP), we might want to identify the element corresponding to the LCP for the page. If the LCP element is an image, knowing the URL of the image resource can help us locate the asset we need to optimize.

Pinpointing the biggest contributor to the Web Vitals score, aka [attribution ↗](#), allows us to obtain more in-depth information like entries for [PerformanceEventTiming ↗](#), [PerformanceNavigationTiming ↗](#) and [PerformanceResourceTiming ↗](#).

Attribution is disabled by default in Next.js but can be enabled **per metric** by specifying the following in `next.config.js`.

`JS next.config.js`

```
1 experimental: {
2   webVitalsAttribution: ['CLS', 'LCP'];
3 }
```

Valid attribution values are all `web-vitals` metrics specified in the [NextWebVitalsMetric ↗](#) type.

> Menu

Pages Router > ... > next.config.js Options > webpack

Custom Webpack Config

Note: changes to webpack config are not covered by semver so proceed at your own risk

Before continuing to add custom webpack configuration to your application make sure Next.js doesn't already support your use-case:

- [CSS imports](#)
- [CSS modules](#)
- [Sass/SCSS imports](#)
- [Sass/SCSS modules](#)
- [preact ↗](#)
- [Customizing babel configuration](#)

Some commonly asked for features are available as plugins:

- [@next/mdx ↗](#)
- [@next/bundle-analyzer ↗](#)

In order to extend our usage of `webpack`, you can define a function that extends its config inside `next.config.js`, like so:

js next.config.js



```
1 module.exports = {
2   webpack: (
3     config,
4     { buildId, dev, isServer, defaultLoaders, nextRuntime, webpack },
5   ) => {
6     // Important: return the modified config
7     return config;
8   },
9 };
```

The `webpack` function is executed twice, once for the server and once for the client. This allows you to distinguish between client and server configuration using the `isServer` property.

The second argument to the `webpack` function is an object with the following properties:

- `buildId` : `String` - The build id, used as a unique identifier between builds
- `dev` : `Boolean` - Indicates if the compilation will be done in development
- `isServer` : `Boolean` - It's `true` for server-side compilation, and `false` for client-side compilation
- `nextRuntime` : `String | undefined` - The target runtime for server-side compilation; either `"edge"` or `"nodejs"`, it's `undefined` for client-side compilation.
- `defaultLoaders` : `Object` - Default loaders used internally by Next.js:
 - `babel` : `Object` - Default `babel-loader` configuration

Example usage of `defaultLoaders.babel`:

```
1 // Example config for adding a loader that depends on babel-loader
2 // This source was taken from the @next-mdx plugin source:
3 // https://github.com/vercel/next.js/tree/canary/packages/next-mdx
4 module.exports = {
5   webpack: (config, options) => {
6     config.module.rules.push({
7       test: /\.mdx/,
8       use: [
9         options.defaultLoaders.babel,
10        {
11          loader: '@mdx-js/loader',
12          options: pluginOptions.options,
13        },
14      ],
15    });
16
17    return config;
18  },
19};
```

nextRuntime

Notice that `isServer` is `true` when `nextRuntime` is `"edge"` or `"nodejs"`, `nextRuntime "edge"` is currently for middleware and Server Components in edge runtime only.

> Menu

Pages Router > Building Your Application

Building Your Application

Next.js provides the building blocks to create flexible, full-stack web applications. The guides in **Building Your Application** explain how to use these features and how to customize your application's behavior.

The sections and pages are organized sequentially, from basic to advanced, so you can follow them step-by-step when building your Next.js application. However, you can read them in any order or skip to the pages that apply to your use case.

If you're new to Next.js, we recommend starting with the [Routing](#), [Rendering](#), [Data Fetching](#) and [Styling](#) sections, as they introduce the fundamental Next.js and web concepts to help you get started. Then, you can dive deeper into the other sections such as [Optimizing](#) and [Configuring](#). Finally, once you're ready, checkout the [Deploying](#) and [Upgrading](#) sections.

Routing

Learn the fundamentals of routing for front-end applications with the Pages Router.

Rendering

Learn the fundamentals of rendering in React and Next.js.

Data Fetching

Next.js allows you to fetch data in multiple ways, with pre-rendering, server-side rendering or static-site generation, and incremental static regeneration. Learn how to manage your application data in Next.js.

Styling

Learn the different ways you can style your Next.js application.

Optimizing

Optimize your Next.js application for best performance and user experience.

Configuring

Learn how to configure your Next.js application.

Deploying

Learn how to deploy your Next.js app to production, either managed or self-hosted.

Upgrading

Learn how to upgrade to the latest versions of Next.js.

> Menu

Pages Router > Building Your Application > Configuring

Configuring

Next.js allows you to customize your project to meet specific requirements. This includes integrations with TypeScript, ESLint, and more, as well as internal configuration options such as Absolute Imports and Environment Variables.

TypeScript

Next.js provides a TypeScript-first development experience for building your React application.

ESLint

Next.js reports ESLint errors and warnings during builds by default. Learn how to opt-out of this behavior [here](#).

Environment Variables

Learn to add and access environment variables in your Next.js application.

Absolute Imports and Module Path Aliases

Configure module path aliases that allow you to remap certain import paths.

src Directory

Save pages under the `src` directory as an alternative to the root `pages` directory.

MDX

Learn how to configure MDX to write JSX in your markdown files.

AMP

With minimal config, and without leaving React, you can start adding AMP and improve the performance and speed of your pages.

Babel

Extend the babel preset added by Next.js with your own configs.

PostCSS

Extend the PostCSS config and plugins added by Next.js with your own.

Custom Server

Start a Next.js app programmatically using a custom server.

Draft Mode

Next.js has draft mode to toggle between static and dynamic pages. You can learn how it works with Pages Router.

Error Handling

Handle errors in your Next.js app.

Debugging

Learn how to debug your Next.js application with VS Code or Chrome DevTools.

Preview Mode (Legacy)

Next.js has the preview mode for statically generated pages. You can learn how it works here.

> Menu

Pages Router > ... > Configuring > Absolute Imports and Module Path Aliases

Absolute Imports and Module Path Aliases

► Examples

Next.js has in-built support for the `"paths"` and `"baseUrl"` options of `tsconfig.json` and `jsconfig.json` files.

These options allow you to alias project directories to absolute paths, making it easier to import modules.

For example:

```
1 // before
2 import { Button } from '../../../../../components/button';
3
4 // after
5 import { Button } from '@/components/button';
```

Good to know: `create-next-app` will prompt to configure these options for you.

Absolute Imports

The `baseUrl` configuration option allows you to import directly from the root of the project.

An example of this configuration:

tsconfig.json / jsconfig.json

```
1 {
2   "compilerOptions": {
3     "baseUrl": "."
4   }
5 }
```

TS components/button.tsx

```
1 export default function Button() {
2   return <button>Click me</button>;
3 }
```

TS app/page.tsx

```
1 import Button from 'components/button';
2
3 export default function HomePage() {
4   return (
5     <>
6       <h1>Hello World</h1>
7       <Button />
8     </>
9   );
10 }
```

Module Aliases

In addition to configuring the `baseUrl` path, you can use the `"paths"` option to "alias" module paths.

For example, the following configuration maps `@/components/*` to `components/*`:

tsconfig.json or jsconfig.json

```
1 {
2   "compilerOptions": {
3     "baseUrl": ".",
4     "paths": {
5       "@/components/*": [ "components/*" ]
6     }
7   }
8 }
```

TS components/button.tsx

```
1 export default function Button() {
2   return <button>Click me</button>;
3 }
```

```
1 import Button from '@/components/button';
2
3 export default function HomePage() {
4   return (
5     <>
6       <h1>Hello World</h1>
7       <Button />
8     </>
9   );
10 }
```

> Menu

Pages Router > ... > Configuring > AMP

AMP

► Examples

With Next.js you can turn any React page into an AMP page, with minimal config, and without leaving React.

You can read more about AMP in the official [amp.dev ↗](#) site.

Enabling AMP

To enable AMP support for a page, and to learn more about the different AMP configs, read the [API documentation for `next/amp`](#).

Caveats

- Only CSS-in-JS is supported. [CSS Modules](#) aren't supported by AMP pages at the moment. You can [contribute CSS Modules support to Next.js ↗](#).
-

Adding AMP Components

The AMP community provides [many components ↗](#) to make AMP pages more interactive. Next.js will automatically import all components used on a page and there is no need to manually import AMP component scripts:

```
1 export const config = { amp: true };  
2
```

```

3  function MyAmpPage() {
4    const date = new Date();
5
6    return (
7      <div>
8        <p>Some time: {date.toJSON()}</p>
9        <amp-timeago
10          width="0"
11          height="15"
12          datetime={date.toJSON()}
13          layout="responsive"
14        >
15        .
16        </amp-timeago>
17      </div>
18    );
19  }
20
21 export default MyAmpPage;

```

The above example uses the [amp-timeago](#) component.

By default, the latest version of a component is always imported. If you want to customize the version, you can use `next/head`, as in the following example:

```

1 import Head from 'next/head';
2
3 export const config = { amp: true };
4
5 function MyAmpPage() {
6   const date = new Date();
7
8   return (
9     <div>
10       <Head>
11         <script
12           async
13           key="amp-timeago"
14           custom-element="amp-timeago"
15           src="https://cdn.ampproject.org/v0/amp-timeago-0.1.js"
16         />
17       </Head>
18
19       <p>Some time: {date.toJSON()}</p>
20       <amp-timeago
21         width="0"
22         height="15"
23         datetime={date.toJSON()}
24         layout="responsive"
25       >
26       .
27       </amp-timeago>
28     </div>

```

```
29   );
30 }
31
32 export default MyAmpPage;
```

AMP Validation

AMP pages are automatically validated with [amphtml-validator](#) during development. Errors and warnings will appear in the terminal where you started Next.js.

Pages are also validated during [Static HTML export](#) and any warnings / errors will be printed to the terminal. Any AMP errors will cause the export to exit with status code `1` because the export is not valid AMP.

Custom Validators

You can set up custom AMP validator in `next.config.js` as shown below:

```
1 module.exports = {
2   amp: {
3     validator: './custom_validator.js',
4   },
5 };
```

Skip AMP Validation

To turn off AMP validation add the following code to `next.config.js`

```
1 experimental: {
2   amp: {
3     skipValidation: true;
4   }
5 }
```

AMP in Static HTML Export

When using [Static HTML export](#) statically prerender pages, Next.js will detect if the page supports AMP and change the exporting behavior based on that.

For example, the hybrid AMP page `pages/about.js` would output:

- `out/about.html` - HTML page with client-side React runtime

- `out/about.amp.html` - AMP page

And if `pages/about.js` is an AMP-only page, then it would output:

- `out/about.html` - Optimized AMP page

Next.js will automatically insert a link to the AMP version of your page in the HTML version, so you don't have to, like so:

```
<link rel="amphtml" href="/about.amp.html" />
```

And the AMP version of your page will include a link to the HTML page:

```
<link rel="canonical" href="/about" />
```

When `trailingSlash` is enabled the exported pages for `pages/about.js` would be:

- `out/about/index.html` - HTML page
- `out/about.amp/index.html` - AMP page

TypeScript

AMP currently doesn't have built-in types for TypeScript, but it's in their roadmap ([#13791 ↗](#)).

As a workaround you can manually create a file called `amp.d.ts` inside your project and add these [custom types](#) ↗.

> Menu

Pages Router > ... > Configuring > Babel

Babel

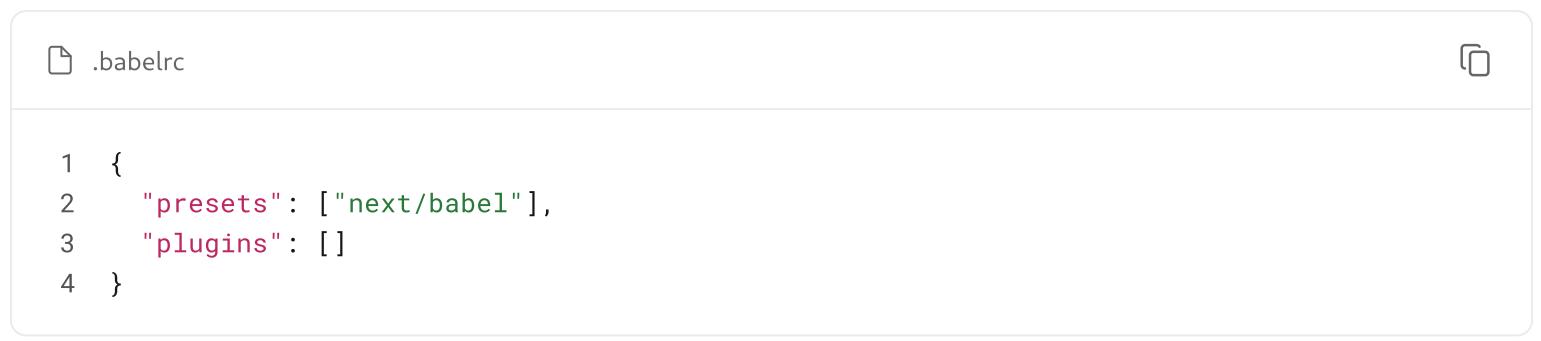
► Examples

Next.js includes the `next/babel` preset to your app, which includes everything needed to compile React applications and server-side code. But if you want to extend the default Babel configs, it's also possible.

Adding Presets and Plugins

To start, you only need to define a `.babelrc` file (or `babel.config.js`) in the root directory of your project. If such a file is found, it will be considered as the *source of truth*, and therefore it needs to define what Next.js needs as well, which is the `next/babel` preset.

Here's an example `.babelrc` file:

A screenshot of a code editor showing a file named ".babelrc". The file contains the following JSON configuration:

```
1  {
2    "presets": ["next/babel"],
3    "plugins": []
4 }
```

The code editor has standard UI elements like a file icon, a save icon, and a scroll bar.

You can [take a look at this file ↗](#) to learn about the presets included by `next/babel`.

To add presets/plugins **without configuring them**, you can do it this way:

A screenshot of a code editor showing a file named ".babelrc" with the following JSON configuration:

```
1  {
2    "presets": ["next/babel"],
3    "plugins": ["@babel/plugin-proposal-do-expressions"]
4 }
```

The code editor has standard UI elements like a file icon, a save icon, and a scroll bar.

Customizing Presets and Plugins

To add presets/plugins **with custom configuration**, do it on the `next/babel` preset like so:

.babelrc

```
1  {
2    "presets": [
3      [
4        "next/babel",
5        {
6          "preset-env": {},
7          "transform-runtime": {},
8          "styled-jsx": {},
9          "class-properties": {}
10         }
11       ]
12     ],
13     "plugins": []
14 }
```



To learn more about the available options for each config, visit babel's [documentation](#) site.

Good to know:

- Next.js uses the **current** Node.js version for server-side compilations.
- The `modules` option on `"preset-env"` should be kept to `false`, otherwise webpack code splitting is turned off.

> Menu

Pages Router > ... > Configuring > Custom Server

Custom Server

► Examples

By default, Next.js includes its own server with `next start`. If you have an existing backend, you can still use it with Next.js (this is not a custom server). A custom Next.js server allows you to start a server 100% programmatically in order to use custom server patterns. Most of the time, you will not need this - but it's available for complete customization.

Note: A custom server **cannot** be deployed on [Vercel ↗](#).

Before deciding to use a custom server, please keep in mind that it should only be used when the integrated router of Next.js can't meet your app requirements. A custom server will remove important performance optimizations, like **serverless functions** and [Automatic Static Optimization](#).

Take a look at the following example of a custom server:

JS server.js



```
1 const { createServer } = require('http');
2 const { parse } = require('url');
3 const next = require('next');
4
5 const dev = process.env.NODE_ENV !== 'production';
6 const hostname = 'localhost';
7 const port = 3000;
8 // when using middleware `hostname` and `port` must be provided below
9 const app = next({ dev, hostname, port });
10 const handle = app.getRequestHandler();
11
12 app.prepare().then(() => {
13   createServer(async (req, res) => {
14     try {
15       // Be sure to pass `true` as the second argument to `url.parse`.
16       // This tells it to parse the query portion of the URL.
17       const parsedUrl = parse(req.url, true);
18       const { pathname, query } = parsedUrl;
```

```

19
20     if (pathname === '/a') {
21         await app.render(req, res, '/a', query);
22     } else if (pathname === '/b') {
23         await app.render(req, res, '/b', query);
24     } else {
25         await handle(req, res, parsedUrl);
26     }
27 } catch (err) {
28     console.error('Error occurred handling', req.url, err);
29     res.statusCode = 500;
30     res.end('internal server error');
31 }
32 })
33 .once('error', (err) => {
34     console.error(err);
35     process.exit(1);
36 })
37 .listen(port, () => {
38     console.log(`Ready on http://${hostname}:${port}`);
39 });
40 });

```

`server.js` doesn't go through babel or webpack. Make sure the syntax and sources this file requires are compatible with the current node version you are running.

To run the custom server you'll need to update the `scripts` in `package.json` like so:

```

1 "scripts": {
2     "dev": "node server.js",
3     "build": "next build",
4     "start": "NODE_ENV=production node server.js"
5 }

```

The custom server uses the following import to connect the server with the Next.js application:

```

1 const next = require('next');
2 const app = next({});

```

The above `next` import is a function that receives an object with the following options:

- `dev : Boolean` - Whether or not to launch Next.js in dev mode. Defaults to `false`
- `dir : String` - Location of the Next.js project. Defaults to `'.'`

- `quiet` : `Boolean` - Hide error messages containing server information. Defaults to `false`
- `conf` : `object` - The same object you would use in `next.config.js`. Defaults to `{}`

The returned `app` can then be used to let Next.js handle requests as required.

Disabling file-system routing

By default, `Next` will serve each file in the `pages` folder under a pathname matching the filename. If your project uses a custom server, this behavior may result in the same content being served from multiple paths, which can present problems with SEO and UX.

To disable this behavior and prevent routing based on files in `pages`, open `next.config.js` and disable the `useFileSystemPublicRoutes` config:

`JS` `next.config.js`

```
1 module.exports = {  
2   useFileSystemPublicRoutes: false,  
3 };
```

Note that `useFileSystemPublicRoutes` disables filename routes from SSR; client-side routing may still access those paths. When using this option, you should guard against navigation to routes you do not want programmatically.

You may also wish to configure the client-side router to disallow client-side redirects to filename routes; for that refer to `router.beforePopState`.

> Menu

Pages Router > ... > Configuring > Debugging

Debugging

This documentation explains how you can debug your Next.js frontend and backend code with full source maps support using either the [VS Code debugger ↗](#) or [Chrome DevTools ↗](#).

Any debugger that can attach to Node.js can also be used to debug a Next.js application. You can find more details in the Node.js [Debugging Guide ↗](#).

Debugging with VS Code

Create a file named `.vscode/launch.json` at the root of your project with the following content:

```
1  {
2      "version": "0.2.0",
3      "configurations": [
4          {
5              "name": "Next.js: debug server-side",
6              "type": "node-terminal",
7              "request": "launch",
8              "command": "npm run dev"
9          },
10         {
11             "name": "Next.js: debug client-side",
12             "type": "chrome",
13             "request": "launch",
14             "url": "http://localhost:3000"
15         },
16         {
17             "name": "Next.js: debug full stack",
18             "type": "node-terminal",
19             "request": "launch",
20             "command": "npm run dev",
21             "serverReadyAction": {
22                 "pattern": "started server on .+, url: (https?:\/\/.+)",
23                 "uriFormat": "%s",
24                 "action": "debugWithChrome"
25             }
26         }
27     }
28 }
```

```
26      }
27  ]
28 }
```

`npm run dev` can be replaced with `yarn dev` if you're using Yarn. If you're [changing the port number](#) your application starts on, replace the `3000` in `http://localhost:3000` with the port you're using instead.

Now go to the Debug panel (`Ctrl+Shift+D` on Windows/Linux, `⌃+⌘+D` on macOS), select a launch configuration, then press `F5` or select **Debug: Start Debugging** from the Command Palette to start your debugging session.

Using the Debugger in Jetbrains WebStorm

Click the drop down menu listing the runtime configuration, and click `Edit Configurations...`. Create a `Javascript Debug` debug configuration with `http://localhost:3000` as the URL. Customize to your liking (e.g. Browser for debugging, store as project file), and click `OK`. Run this debug configuration, and the selected browser should automatically open. At this point, you should have 2 applications in debug mode: the NextJS node application, and the client/ browser application.

Debugging with Chrome DevTools

Client-side code

Start your development server as usual by running `next dev`, `npm run dev`, or `yarn dev`. Once the server starts, open `http://localhost:3000` (or your alternate URL) in Chrome. Next, open Chrome's Developer Tools (`Ctrl+Shift+J` on Windows/Linux, `⌃+⌘+I` on macOS), then go to the **Sources** tab.

Now, any time your client-side code reaches a `debugger` statement, code execution will pause and that file will appear in the debug area. You can also press `Ctrl+P` on Windows/Linux or `⌘+P` on macOS to search for a file and set breakpoints manually. Note that when searching here, your source files will have paths starting with `webpack:///_N_E/./`.

Server-side code

To debug server-side Next.js code with Chrome DevTools, you need to pass the `--inspect` flag to the underlying Node.js process:

>_ Terminal

```
NODE_OPTIONS='--inspect' next dev
```

If you're using `npm run dev` or `yarn dev` then you should update the `dev` script on your `package.json`:

```
"dev": "NODE_OPTIONS='--inspect' next dev"
```

Launching the Next.js dev server with the `--inspect` flag will look something like this:

>_ Terminal

```
1 Debugger listening on ws://127.0.0.1:9229/0cf90313-350d-4466-a748-cd60f4e47c95
2 For help, see: https://nodejs.org/en/docs/inspector
3 ready - started server on 0.0.0.0:3000, url: http://localhost:3000
```

Be aware that running `NODE_OPTIONS='--inspect' npm run dev` or `NODE_OPTIONS='--inspect' yarn dev` won't work. This would try to start multiple debuggers on the same port: one for the npm/yarn process and one for Next.js. You would then get an error like `Starting inspector on 127.0.0.1:9229 failed: address already in use` in your console.

Once the server starts, open a new tab in Chrome and visit `chrome://inspect`, where you should see your Next.js application inside the **Remote Target** section. Click **inspect** under your application to open a separate DevTools window, then go to the **Sources** tab.

Debugging server-side code here works much like debugging client-side code with Chrome DevTools, except that when you search for files here with `Ctrl+P` or `⌘+P`, your source files will have paths starting with `webpack:///{application-name}/./` (where `{application-name}` will be replaced with the name of your application according to your `package.json` file).

Debugging on Windows

Windows users may run into an issue when using `NODE_OPTIONS='--inspect'` as that syntax is not supported on Windows platforms. To get around this, install the [cross-env](#) package as a development dependency (`-D` with `npm` and `yarn`) and replace the `dev` script with the following.

```
"dev": "cross-env NODE_OPTIONS='--inspect' next dev",
```

`cross-env` will set the `NODE_OPTIONS` environment variable regardless of which platform you are on (including Mac, Linux, and Windows) and allow you to debug consistently across devices and operating

systems.

Note: Ensure Windows Defender is disabled on your machine. This external service will check *every file read*, which has been reported to greatly increase Fast Refresh time with `next dev`. This is a known issue, not related to Next.js, but it does affect Next.js development.

More information

To learn more about how to use a JavaScript debugger, take a look at the following documentation:

- [Node.js debugging in VS Code: Breakpoints ↗](#)
- [Chrome DevTools: Debug JavaScript ↗](#)

> Menu

Pages Router > ... > Configuring > Draft Mode

Draft Mode

In the [Pages documentation](#) and the [Data Fetching documentation](#), we talked about how to pre-render a page at build time (**Static Generation**) using `getStaticProps` and `getStaticPaths`.

Static Generation is useful when your pages fetch data from a headless CMS. However, it's not ideal when you're writing a draft on your headless CMS and want to view the draft immediately on your page. You'd want Next.js to render these pages at **request time** instead of build time and fetch the draft content instead of the published content. You'd want Next.js to bypass Static Generation only for this specific case.

Next.js has a feature called **Draft Mode** which solves this problem. Here are instructions on how to use it.

Step 1: Create and access the API route

Take a look at the [API Routes documentation](#) first if you're not familiar with Next.js API Routes.

First, create the **API route**. It can have any name - e.g. `pages/api/draft.ts`

In this API route, you need to call `setDraftMode` on the response object.

```
1 export default function handler(req, res) {  
2   // ...  
3   res.setDraftMode({ enable: true });  
4   // ...  
5 }
```

This will set a **cookie** to enable draft mode. Subsequent requests containing this cookie will trigger **Draft Mode** changing the behavior for statically generated pages (more on this later).

You can test this manually by creating an API route like below and accessing it from your browser manually:



```
1 // simple example for testing it manually from your browser.
2 export default function handler(req, res) {
3   res.setDraftMode({ enable: true });
4   res.end('Draft mode is enabled');
5 }
```

If you open your browser's developer tools and visit `/api/draft`, you'll notice a `Set-Cookie` response header with a cookie named `_prerender_bypass`.

Securely accessing it from your Headless CMS

In practice, you'd want to call this API route *securely* from your headless CMS. The specific steps will vary depending on which headless CMS you're using, but here are some common steps you could take.

These steps assume that the headless CMS you're using supports setting **custom draft URLs**. If it doesn't, you can still use this method to secure your draft URLs, but you'll need to construct and access the draft URL manually.

First, you should create a **secret token string** using a token generator of your choice. This secret will only be known by your Next.js app and your headless CMS. This secret prevents people who don't have access to your CMS from accessing draft URLs.

Second, if your headless CMS supports setting custom draft URLs, specify the following as the draft URL. This assumes that your draft API route is located at `pages/api/draft.ts`.

>_ Terminal



```
https://<your-site>/api/draft?secret=<token>&slug=<path>
```

- `<your-site>` should be your deployment domain.
- `<token>` should be replaced with the secret token you generated.
- `<path>` should be the path for the page that you want to view. If you want to view `/posts/foo`, then you should use `&slug=/posts/foo`.

Your headless CMS might allow you to include a variable in the draft URL so that `<path>` can be set dynamically based on the CMS's data like so: `&slug=/posts/{entry.fields.slug}`

Finally, in the draft API route:

- Check that the secret matches and that the `slug` parameter exists (if not, the request should fail).

- Call `res.setDraftMode`.
- Then redirect the browser to the path specified by `slug`. (The following example uses a [307 redirect ↗](#)).

```

1  export default async (req, res) => {
2    // Check the secret and next parameters
3    // This secret should only be known to this API route and the CMS
4    if (req.query.secret !== 'MY_SECRET_TOKEN' || !req.query.slug) {
5      return res.status(401).json({ message: 'Invalid token' });
6    }
7
8    // Fetch the headless CMS to check if the provided `slug` exists
9    // getPostBySlug would implement the required fetching logic to the headless CMS
10   const post = await getPostBySlug(req.query.slug);
11
12   // If the slug doesn't exist prevent draft mode from being enabled
13   if (!post) {
14     return res.status(401).json({ message: 'Invalid slug' });
15   }
16
17   // Enable Draft Mode by setting the cookie
18   res.setDraftMode({ enable: true });
19
20   // Redirect to the path from the fetched post
21   // We don't redirect to req.query.slug as that might lead to open redirect vulnerability
22   res.redirect(post.slug);
23 };

```

If it succeeds, then the browser will be redirected to the path you want to view with the draft mode cookie.

Step 2: Update `getStaticProps`

The next step is to update `getStaticProps` to support draft mode.

If you request a page which has `getStaticProps` with the cookie set (via `res.setDraftMode`), then `getStaticProps` will be called at **request time** (instead of at build time).

Furthermore, it will be called with a `context` object where `context.draftMode` will be `true`.

```

1  export async function getStaticProps(context) {
2    if (context.draftMode) {
3      // dynamic data
4    }
5  }

```

We used `res.setDraftMode` in the draft API route, so `context.draftMode` will be `true`.

If you're also using `getStaticPaths`, then `context.params` will also be available.

Fetch draft data

You can update `getStaticProps` to fetch different data based on `context.draftMode`.

For example, your headless CMS might have a different API endpoint for draft posts. If so, you can modify the API endpoint URL like below:

```
1 export async function getStaticProps(context) {  
2   const url = context.draftMode  
3     ? 'https://draft.example.com'  
4     : 'https://production.example.com';  
5   const res = await fetch(url);  
6   // ...  
7 }
```

That's it! If you access the draft API route (with `secret` and `slug`) from your headless CMS or manually, you should now be able to see the draft content. And if you update your draft without publishing, you should be able to view the draft.

Set this as the draft URL on your headless CMS or access manually, and you should be able to see the draft.

>_ Terminal



```
https://<your-site>/api/draft?secret=<token>&slug=<path>
```

More Details

Clear the Draft Mode cookie

By default, the Draft Mode session ends when the browser is closed.

To clear the Draft Mode cookie manually, create an API route that calls

```
setDraftMode({ enable: false }):
```

ts pages/api/disable-draft.ts



```
1 export default function handler(req, res) {  
2   res.setDraftMode({ enable: false });  
3 }
```

Then, send a request to `/api/disable-draft` to invoke the API Route. If calling this route using `next/link`, you must pass `prefetch={false}` to prevent accidentally deleting the cookie on prefetch.

Works with `getServerSideProps`

Draft Mode works with `getServerSideProps` as well. It will also be available on the `context` object containing `draftMode`

Works with API Routes

API Routes will have access to `draftMode` on the request object. For example:

```
1 export default function myApiRoute(req, res) {  
2   if (req.draftMode) {  
3     // get draft data  
4   }  
5 }
```

Unique per `next build`

A new bypass cookie value will be generated each time you run `next build`.

This ensures that the bypass cookie can't be guessed.

Note: To test Draft Mode locally over HTTP, your browser will need to allow third-party cookies and local storage access.

> Menu

Pages Router > ... > Configuring > Environment Variables

Environment Variables

► Examples

Next.js comes with built-in support for environment variables, which allows you to do the following:

- Use `.env.local` to load environment variables
- Expose environment variables to the browser by prefixing with `NEXT_PUBLIC_`

Loading Environment Variables

Next.js has built-in support for loading environment variables from `.env.local` into `process.env`.

 .env.local

```
1 DB_HOST=localhost  
2 DB_USER=myuser  
3 DB_PASS=mypassword
```

This loads `process.env.DB_HOST`, `process.env.DB_USER`, and `process.env.DB_PASS` into the Node.js environment automatically allowing you to use them in [Next.js data fetching methods](#) and [API routes](#).

For example, using `getStaticProps`:

 pages/index.js

```
1 export async function getStaticProps() {  
2   const db = await myDB.connect({  
3     host: process.env.DB_HOST,  
4     username: process.env.DB_USER,  
5     password: process.env.DB_PASS,  
6   });  
7   // ...
```

Referencing Other Variables

Next.js will automatically expand variables that use `$` to reference other variables e.g. `$VARIABLE` inside of your `.env*` files. This allows you to reference other secrets. For example:

 .env

```
1 HOSTNAME=localhost
2 PORT=8080
3 HOST=http://$HOSTNAME:$PORT
```



In the above example, `process.env.HOST` would be set to `http://localhost:8080`.

Note: If you need to use variable with a `$` in the actual value, it needs to be escaped e.g. `\$`.

Exposing Environment Variables to the Browser

By default environment variables are only available in the Node.js environment, meaning they won't be exposed to the browser.

In order to expose a variable to the browser you have to prefix the variable with `NEXT_PUBLIC_`. For example:

>_ Terminal



```
NEXT_PUBLIC_ANALYTICS_ID=abcdefghijklm
```

This loads `process.env.NEXT_PUBLIC_ANALYTICS_ID` into the Node.js environment automatically, allowing you to use it anywhere in your code. The value will be inlined into JavaScript sent to the browser because of the `NEXT_PUBLIC_` prefix. This inlining occurs at build time, so your various `NEXT_PUBLIC_` envs need to be set when the project is built.

 pages/index.js



```
1 import setupAnalyticsService from '../lib/my-analytics-service';
2
```

```
3 // 'NEXT_PUBLIC_ANALYTICS_ID' can be used here as it's prefixed by 'NEXT_PUBLIC_'.  
4 // It will be transformed at build time to `setupAnalyticsService('abcdefgijk')`.  
5 setupAnalyticsService(process.env.NEXT_PUBLIC_ANALYTICS_ID);  
6  
7 function HomePage() {  
8   return <h1>Hello World</h1>;  
9 }  
10  
11 export default HomePage;
```

Note that dynamic lookups will *not* be inlined, such as:

```
1 // This will NOT be inlined, because it uses a variable  
2 const varName = 'NEXT_PUBLIC_ANALYTICS_ID';  
3 setupAnalyticsService(process.env[varName]);  
4  
5 // This will NOT be inlined, because it uses a variable  
6 const env = process.env;  
7 setupAnalyticsService(env.NEXT_PUBLIC_ANALYTICS_ID);
```

Default Environment Variables

In general only one `.env.local` file is needed. However, sometimes you might want to add some defaults for the `development` (`next dev`) or `production` (`next start`) environment.

Next.js allows you to set defaults in `.env` (all environments), `.env.development` (development environment), and `.env.production` (production environment).

`.env.local` always overrides the defaults set.

Note: `.env`, `.env.development`, and `.env.production` files should be included in your repository as they define defaults. `.env*.local` should be added to `.gitignore`, as those files are intended to be ignored. `.env.local` is where secrets can be stored.

Environment Variables on Vercel

When deploying your Next.js application to [Vercel](#), Environment Variables can be configured [in the Project Settings](#).

All types of Environment Variables should be configured there. Even Environment Variables used in Development – which can be [downloaded onto your local device ↗](#) afterwards.

If you've configured [Development Environment Variables ↗](#) you can pull them into a `.env.local` for usage on your local machine using the following command:

>_ Terminal

```
vercel env pull .env.local
```

Test Environment Variables

Apart from `development` and `production` environments, there is a 3rd option available: `test`. In the same way you can set defaults for development or production environments, you can do the same with a `.env.test` file for the `testing` environment (though this one is not as common as the previous two).

Next.js will not load environment variables from `.env.development` or `.env.production` in the `testing` environment.

This one is useful when running tests with tools like `jest` or `cypress` where you need to set specific environment vars only for testing purposes. Test default values will be loaded if `NODE_ENV` is set to `test`, though you usually don't need to do this manually as testing tools will address it for you.

There is a small difference between `test` environment, and both `development` and `production` that you need to bear in mind: `.env.local` won't be loaded, as you expect tests to produce the same results for everyone. This way every test execution will use the same env defaults across different executions by ignoring your `.env.local` (which is intended to override the default set).

Note: similar to Default Environment Variables, `.env.test` file should be included in your repository, but `.env.test.local` shouldn't, as `.env*.local` are intended to be ignored through `.gitignore`.

While running unit tests you can make sure to load your environment variables the same way Next.js does by leveraging the `loadEnvConfig` function from the `@next/env` package.

```
1 // The below can be used in a Jest global setup file or similar for your testing set-up
2 import { loadEnvConfig } from '@next/env';
3
4 export default async () => {
5   const projectDir = process.cwd();
6   loadEnvConfig(projectDir);
```

Environment Variable Load Order

Environment variables are looked up in the following places, in order, stopping once the variable is found.

1. process.env
2. .env.\$(NODE_ENV).local
3. .env.local (Not checked when NODE_ENV is test.)
4. .env.\$(NODE_ENV)
5. .env

For example, if NODE_ENV is development and you define a variable in both .env.development.local and .env, the value in .env.development.local will be used.

Note: The allowed values for NODE_ENV are production, development and test.

Good to know

- If you are using a /src directory, env.* files should remain in the root of your project.

> Menu

Pages Router > ... > Configuring > Error Handling

Error Handling

This documentation explains how you can handle development, server-side, and client-side errors.

Handling Errors in Development

When there is a runtime error during the development phase of your Next.js application, you will encounter an **overlay**. It is a modal that covers the webpage. It is **only** visible when the development server runs using `next dev` via `pnpm dev`, `npm run dev`, or `yarn dev` and will not be shown in production. Fixing the error will automatically dismiss the overlay.

Here is an example of an overlay:

Failed to compile

```
./pages/index.js:5:0
Module not found: Can't resolve '../components/header'
3 | import styles from '../styles/Home.module.css';
4 |
> 5 | import Header from '../components/header';
6 |
7 | export default function Home() {
8 |   return (
```

<https://nextjs.org/docs/messages/module-not-found>

This error occurred during the build process and can only be dismissed by fixing the error.

Handling Server Errors

Next.js provides a static 500 page by default to handle server-side errors that occur in your application. You can also [customize this page](#) by creating a `pages/500.js` file.

Having a 500 page in your application does not show specific errors to the app user.

You can also use [404 page](#) to handle specific runtime error like `file not found`.

Handling Client Errors

React [Error Boundaries](#) is a graceful way to handle a JavaScript error on the client so that the other parts of the application continue working. In addition to preventing the page from crashing, it allows you to provide a custom fallback component and even log error information.

To use Error Boundaries for your Next.js application, you must create a class component `ErrorBoundary` and wrap the `Component` prop in the `pages/_app.js` file. This component will be responsible to:

- Render a fallback UI after an error is thrown
- Provide a way to reset the Application's state
- Log error information

You can create an `ErrorBoundary` class component by extending `React.Component`. For example:

```
1  class ErrorBoundary extends React.Component {
2      constructor(props) {
3          super(props);
4
5          // Define a state variable to track whether is an error or not
6          this.state = { hasError: false };
7      }
8      static getDerivedStateFromError(error) {
9          // Update state so the next render will show the fallback UI
10
11         return { hasError: true };
12     }
13     componentDidCatch(error, errorInfo) {
14         // You can use your own error logging service here
15         console.log({ error, errorInfo });
16     }
17     render() {
18         // Check if the error is thrown
19         if (this.state.hasError) {
20             // You can render any custom fallback UI
21             return (
22                 <div>
23                     <h2>Oops, there is an error!</h2>
```

```

24         <button
25             type="button"
26             onClick={() => this.setState({ hasError: false })}
27         >
28             Try again?
29         </button>
30     </div>
31 );
32 }
33
34 // Return children components in case of no error
35
36 return this.props.children;
37 }
38 }
39
40 export default ErrorBoundary;

```

The `ErrorBoundary` component keeps track of an `hasError` state. The value of this state variable is a boolean. When the value of `hasError` is `true`, then the `ErrorBoundary` component will render a fallback UI. Otherwise, it will render the children components.

After creating an `ErrorBoundary` component, import it in the `pages/_app.js` file to wrap the `Component` prop in your Next.js application.

```

1 // Import the ErrorBoundary component
2 import ErrorBoundary from '../components/ErrorBoundary';
3
4 function MyApp({ Component, pageProps }) {
5     return (
6         // Wrap the Component prop with ErrorBoundary component
7         <ErrorBoundary>
8             <Component {...pageProps} />
9         </ErrorBoundary>
10    );
11 }
12
13 export default MyApp;

```

You can learn more about [Error Boundaries ↗](#) in React's documentation.

Reporting Errors

To monitor client errors, use a service like [Sentry ↗](#), Bugsnag or Datadog.

> Menu

Pages Router > ... > Configuring > ESLint

ESLint

Next.js provides an integrated [ESLint](#) experience out of the box. Add `next lint` as a script to `package.json`:

 package.json

```
1 "scripts": {  
2   "lint": "next lint"  
3 }
```

Then run `npm run lint` or `yarn lint`:

 Terminal`yarn lint`

If you don't already have ESLint configured in your application, you will be guided through the installation and configuration process.

 Terminal`yarn lint`

You'll see a prompt like this:

? How would you like to configure ESLint?

› Base configuration + Core Web Vitals rule-set (recommended) Base configuration None

One of the following three options can be selected:

- **Strict:** Includes Next.js' base ESLint configuration along with a stricter [Core Web Vitals rule-set](#). This is the recommended configuration for developers setting up ESLint for the first time.

```
1 {
2   "extends": "next/core-web-vitals"
3 }
```

- **Base:** Includes Next.js' base ESLint configuration.

```
1 {
2   "extends": "next"
3 }
```

- **Cancel:** Does not include any ESLint configuration. Only select this option if you plan on setting up your own custom ESLint configuration.

If either of the two configuration options are selected, Next.js will automatically install `eslint` and `eslint-config-next` as development dependencies in your application and create an `.eslintrc.json` file in the root of your project that includes your selected configuration.

You can now run `next lint` every time you want to run ESLint to catch errors. Once ESLint has been set up, it will also automatically run during every build (`next build`). Errors will fail the build, while warnings will not.

If you do not want ESLint to run during `next build`, refer to the documentation for [Ignoring ESLint](#).

We recommend using an appropriate [integration](#) to view warnings and errors directly in your code editor during development.

ESLint Config

The default configuration (`eslint-config-next`) includes everything you need to have an optimal out-of-the-box linting experience in Next.js. If you do not have ESLint already configured in your application, we recommend using `next lint` to set up ESLint along with this configuration.

If you would like to use `eslint-config-next` along with other ESLint configurations, refer to the [Additional Configurations](#) section to learn how to do so without causing any conflicts.

Recommended rule-sets from the following ESLint plugins are all used within `eslint-config-next`:

- [eslint-plugin-react](#)

- [eslint-plugin-react-hooks ↗](#)
- [eslint-plugin-next ↗](#)

This will take precedence over the configuration from `next.config.js`.

ESLint Plugin

Next.js provides an ESLint plugin, [eslint-plugin-next ↗](#), already bundled within the base configuration that makes it possible to catch common issues and problems in a Next.js application. The full set of rules is as follows:

- ✓ Enabled in the recommended configuration

Rule	Description
✓ @next/next/google-font-display	Enforce font-display behavior with Google Fonts.
✓ @next/next/google-font-preconnect	Ensure <code>preconnect</code> is used with Google Fonts.
✓ @next/next/inline-script-id	Enforce <code>id</code> attribute on <code>next/script</code> components with inline content.
✓ @next/next/next-script-for-ga	Prefer <code>next/script</code> component when using the inline script for Google Analytics.
✓ @next/next/no-assign-module-variable	Prevent assignment to the <code>module</code> variable.
✓ @next/next/no-before-interactive-script-outside-document	Prevent usage of <code>next/script</code> 's <code>beforeInteractive</code> strategy outside of <code>pages/_document.js</code> .
✓ @next/next/no-css-tags	Prevent manual stylesheet tags.
✓ @next/next/no-document-import-in-page	Prevent importing <code>next/document</code> outside of <code>pages/_document.js</code> .
✓ @next/next/no-duplicate-head	Prevent duplicate usage of <code><Head></code> in <code>pages/_document.js</code> .
✓ @next/next/no-head-element	Prevent usage of <code><head></code> element.
✓ @next/next/no-head-import-in-document	Prevent usage of <code>next/head</code> in <code>pages/_document.js</code> .
✓ @next/next/no-html-link-for-pages	Prevent usage of <code><a></code> elements to navigate to internal Next.js pages.
✓ @next/next/no-img-element	Prevent usage of <code></code> element due to slower LCP and higher bandwidth.

Rule	Description
✓ @next/next/no-page-custom-font	Prevent page-only custom fonts.
✓ @next/next/no-script-component-in-head	Prevent usage of <code>next/script</code> in <code>next/head</code> component.
✓ @next/next/no-styled-jsx-in-document	Prevent usage of <code>styled-jsx</code> in <code>pages/_document.js</code> .
✓ @next/next/no-sync-scripts	Prevent synchronous scripts.
✓ @next/next/no-title-in-document-head	Prevent usage of <code><title></code> with <code>Head</code> component from <code>next/document</code> .
✓ @next/next/no-typos	Prevent common typos in Next.js's data fetching functions
✓ @next/next/no-unwanted-polyfillio	Prevent duplicate polyfills from Polyfill.io.

If you already have ESLint configured in your application, we recommend extending from this plugin directly instead of including `eslint-config-next` unless a few conditions are met. Refer to the [Recommended Plugin Ruleset](#) to learn more.

Custom Settings

`rootDir`

If you're using `eslint-plugin-next` in a project where Next.js isn't installed in your root directory (such as a monorepo), you can tell `eslint-plugin-next` where to find your Next.js application using the `settings` property in your `.eslintrc`:

 `.eslintrc`



```

1  {
2    "extends": "next",
3    "settings": {
4      "next": {
5        "rootDir": "packages/my-app/"
6      }
7    }
8  }

```

`rootDir` can be a path (relative or absolute), a glob (i.e. `"packages/**/*"`), or an array of paths and/or globs.

Linting Custom Directories and Files

By default, Next.js will run ESLint for all files in the `pages/`, `app` (only if the experimental `appDir` feature is enabled), `components/`, `lib/`, and `src/` directories. However, you can specify which directories using the `dirs` option in the `eslint` config in `next.config.js` for production builds:

JS next.config.js

```
1 module.exports = {
2   eslint: {
3     dirs: ['pages', 'utils'], // Only run ESLint on the 'pages' and 'utils' directories
4   },
5 }
```

Similarly, the `--dir` and `--file` flags can be used for `next lint` to lint specific directories and files:

> Terminal

```
next lint --dir pages --dir utils --file bar.js
```

Caching

To improve performance, information of files processed by ESLint are cached by default. This is stored in `.next/cache` or in your defined [build directory](#). If you include any ESLint rules that depend on more than the contents of a single source file and need to disable the cache, use the `--no-cache` flag with `next lint`.

> Terminal

```
next lint --no-cache
```

Disabling Rules

If you would like to modify or disable any rules provided by the supported plugins (`react`, `react-hooks`, `next`), you can directly change them using the `rules` property in your `.eslintrc`:

File .eslintrc

```
1  {
2    "extends": "next",
3    "rules": {
4      "react/no-unesaped-entities": "off",
5      "@next/next/no-page-custom-font": "off"
6    }
7 }
```

Core Web Vitals

The `next/core-web-vitals` rule set is enabled when `next lint` is run for the first time and the **strict** option is selected.

```
1  {
2    "extends": "next/core-web-vitals"
3 }
```

`next/core-web-vitals` updates `eslint-plugin-next` to error on a number of rules that are warnings by default if they affect [Core Web Vitals ↗](#).

The `next/core-web-vitals` entry point is automatically included for new applications built with [Create Next App](#).

Usage With Other Tools

Prettier

ESLint also contains code formatting rules, which can conflict with your existing [Prettier ↗](#) setup. We recommend including [eslint-config-prettier ↗](#) in your ESLint config to make ESLint and Prettier work together.

First, install the dependency:

>_ Terminal

```
1 npm install --save-dev eslint-config-prettier
2
3 yarn add --dev eslint-config-prettier
```

Then, add `prettier` to your existing ESLint config:

```
1  {
2    "extends": ["next", "prettier"]
3 }
```

lint-staged

If you would like to use `next lint` with [lint-staged](#) to run the linter on staged git files, you'll have to add the following to the `.lintstagedrc.js` file in the root of your project in order to specify usage of the `--file` flag.

JS .lintstagedrc.js

```
1 const path = require('path');
2
3 const buildEslintCommand = (filenames) =>
4   `next lint --fix --file ${filenames
5     .map((f) => path.relative(process.cwd(), f))
6     .join(' --file '))`;
7
8 module.exports = {
9   `*.${js,jsx,ts,tsx}`: [buildEslintCommand],
10};
```

Migrating Existing Config

Recommended Plugin Ruleset

If you already have ESLint configured in your application and any of the following conditions are true:

- You have one or more of the following plugins already installed (either separately or through a different config such as `airbnb` or `react-app`):
 - `react`
 - `react-hooks`
 - `jsx-a11y`
 - `import`
- You've defined specific `parserOptions` that are different from how Babel is configured within Next.js (this is not recommended unless you have [customized your Babel configuration](#))

- You have `eslint-plugin-import` installed with Node.js and/or TypeScript [resolvers](#) defined to handle imports

Then we recommend either removing these settings if you prefer how these properties have been configured within `eslint-config-next` [or](#) or extending directly from the Next.js ESLint plugin instead:

```
1 module.exports = {
2   extends: [
3     //...
4     'plugin:@next/next/recommended',
5   ],
6 };
```

The plugin can be installed normally in your project without needing to run `next lint`:

> Terminal

```
1 npm install --save-dev @next/eslint-plugin-next
2
3 yarn add --dev @next/eslint-plugin-next
```

This eliminates the risk of collisions or errors that can occur due to importing the same plugin or parser across multiple configurations.

Additional Configurations

If you already use a separate ESLint configuration and want to include `eslint-config-next`, ensure that it is extended last after other configurations. For example:

```
1 {
2   "extends": ["eslint:recommended", "next"]
3 }
```

The `next` configuration already handles setting default values for the `parser`, `plugins` and `settings` properties. There is no need to manually re-declare any of these properties unless you need a different configuration for your use case.

If you include any other shareable configurations, **you will need to make sure that these properties are not overwritten or modified**. Otherwise, we recommend removing any configurations that share behavior with the `next` configuration or extending directly from the Next.js ESLint plugin as mentioned above.

> Menu

Pages Router > ... > Configuring > MDX

MDX

[Markdown ↗](#) is a lightweight markup language used to format text. It allows you to write using plain text syntax and convert it to structurally valid HTML. It's commonly used for writing content on websites and blogs.

You write...

```
I **love** using [Next.js](https://nextjs.org/)
```

Output:

```
<p>I <strong>love</strong> using <a href="https://nextjs.org/">Next.js</a></p>
```

[MDX ↗](#) is a superset of markdown that lets you write [JSX ↗](#) directly in your markdown files. It is a powerful way to add dynamic interactivity and embed React components within your content.

Next.js can support both local MDX content inside your application, as well as remote MDX files fetched dynamically on the server. The Next.js plugin handles transforming Markdown and React components into HTML, including support for usage in Server Components (default in `app`).

`@next/mdx`

The `@next/mdx` package is configured in the `next.config.js` file at your projects root. **It sources data from local files**, allowing you to create pages with a `.mdx` extension, directly in your `/pages` or `/app` directory.

Getting Started

Install the required packages:

>_ Terminal

```
npm install @next/mdx @mdx-js/loader @mdx-js/react
```

Require the package and configure to support top level `.mdx` pages. The following adds the `options` object key allowing you to pass in any plugins:

```
1 // next.config.js
2
3 const withMDX = require('@next/mdx')({
4   extension: /\.mdx?$/,
5   options: {
6     // If you use remark-gfm, you'll need to use next.config.mjs
7     // as the package is ESM only
8     // https://github.com/remarkjs/remark-gfm#install
9     remarkPlugins: [],
10    rehypePlugins: [],
11    // If you use `MDXProvider`, uncomment the following line.
12    // providerImportSource: "@mdx-js/react",
13  },
14});
15
16 /** @type {import('next').NextConfig} */
17 const nextConfig = {
18   // Configure pageExtensions to include md and mdx
19   pageExtensions: ['ts', 'tsx', 'js', 'jsx', 'md', 'mdx'],
20   // Optionally, add any other Next.js config below
21   reactStrictMode: true,
22 };
23
24 // Merge MDX config with Next.js config
25 module.exports = withMDX(nextConfig);
```

- Create a new MDX page within the `/pages` directory:

```
1 your-project
2   └── pages
3     └── my-mdx-page.mdx
4     └── package.json
```

You can now import a React component directly inside your MDX page:

```
1 import { MyComponent } from 'my-components';
2
3 My MDX page
4
5 This is a list in markdown:
6
7 - One
8 - Two
9 - Three
10
11 Checkout my React component:
12
13 <MyComponent />
```

Remote MDX

If your Markdown or MDX files do *not* live inside your application, you can fetch them dynamically on the server. This is useful for fetching content from a CMS or other data source.

There are two popular community packages for fetching MDX content: [next-mdx-remote](#) ↗ and [contentlayer](#) ↗. For example, the following example uses `next-mdx-remote`:

Note: Please proceed with caution. MDX compiles to JavaScript and is executed on the server. You should only fetch MDX content from a trusted source, otherwise this can lead to remote code execution (RCE).

app/page.tsx

```
1 import { MDXRemote } from 'next-mdx-remote/rsc';
2
3 export default async function Home() {
4   const res = await fetch('https://...');

5   const markdown = await res.text();
6   return <MDXRemote source={markdown} />;
7 }
```

Layouts

To add a layout to your MDX page, create a new component and import it into the MDX page. Then you can wrap the MDX page with your layout component:



```
1 import { MyLayoutComponent } from 'my-components';
2 import HelloWorld from './hello.mdx';
3
4 export const meta = {
5   author: 'Rich Haines',
6 };
7
8 export default Page({ children }) => (
9   <MyLayoutComponent meta={meta}><HelloWorld /></MyLayoutComponent>
10 );
```

Remark and Rehype Plugins

You can optionally provide `remark` and `rehype` plugins to transform the MDX content. For example, you can use `remark-gfm` to support GitHub Flavored Markdown.

Since the `remark` and `rehype` ecosystem is ESM only, you'll need to use `next.config.mjs` as the configuration file.



```
1 /** @type {import('next').NextConfig} */
2 const nextConfig = {
3   experimental: {
4     appDir: true,
5   },
6 };
7
8 const withMDX = require('@next/mdx')({
9   options: {
10     remarkPlugins: [],
11     rehypePlugins: [],
12     // If you use `MDXProvider`, uncomment the following line.
13     // providerImportSource: "@mdx-js/react",
14   },
15 });
16 module.exports = withMDX(nextConfig);
```

Frontmatter is a YAML like key/value pairing that can be used to store data about a page. `@next/mdx` does **not** support frontmatter by default, though there are many solutions for adding frontmatter to your MDX content, such as [gray-matter](#).

To access page metadata with `@next/mdx`, you can export a meta object from within the `.mdx` file:

```
1 export const meta = {  
2   author: 'Rich Haines',  
3 };  
4  
5 # My MDX page
```

Custom Elements

One of the pleasant aspects of using markdown, is that it maps to native `HTML` elements, making writing fast, and intuitive:

```
1 This is a list in markdown:  
2  
3 - One  
4 - Two  
5 - Three
```

The above generates the following `HTML`:

```
1 <p>This is a list in markdown:</p>  
2  
3 <ul>  
4   <li>One</li>  
5   <li>Two</li>  
6   <li>Three</li>  
7 </ul>
```

When you want to style your own elements to give a custom feel to your website or application, you can pass in shortcodes. These are your own custom components that map to `HTML` elements. To do this you use the `MDXProvider` and pass a components object as a prop. Each object key in the components object maps to a `HTML` element name.

To enable you need to specify `providerImportSource: "@mdx-js/react"` in `next.config.js`.

JS next.config.js

```
1 const withMDX = require('@next/mdx')({
2   // ...
3   options: {
4     providerImportSource: '@mdx-js/react',
5   },
6 });
```

Then setup the provider in your page

JS pages/index.js

```
1
2 import { MDXProvider } from '@mdx-js/react'
3 import Image from 'next/image'
4 import { Heading, InlineCode, Pre, Table, Text } from 'my-components'
5
6 const ResponsiveImage = (props) => (
7   <Image alt={props.alt} sizes="100vw" style={{ width: '100%', height: 'auto' }} {...props}
8 )
9
10 const components = {
11   img: ResponsiveImage,
12   h1: Heading.H1,
13   h2: Heading.H2,
14   p: Text,
15   pre: Pre,
16   code: InlineCode,
17 }
18
19 export default function Post(props) {
20   return (
21     <MDXProvider components={components}>
22       <main {...props} />
23     </MDXProvider>
24   )
25 }
```

If you use it across the site you may want to add the provider to `_app.js` so all MDX pages pick up the custom element config.

Deep Dive: How do you transform markdown into HTML?

React does not natively understand Markdown. The markdown plaintext needs to first be transformed into HTML. This can be accomplished with `remark` and `rehype`.

`remark` is an ecosystem of tools around markdown. `rehype` is the same, but for HTML. For example, the following code snippet transforms markdown into HTML:

```
1 import { unified } from 'unified';
2 import remarkParse from 'remark-parse';
3 import remarkRehype from 'remark-rehype';
4 import rehypeSanitize from 'rehype-sanitize';
5 import rehypeStringify from 'rehype-stringify';
6
7 main();
8
9 async function main() {
10   const file = await unified()
11     .use(remarkParse) // Convert into markdown AST
12     .use(remarkRehype) // Transform to HTML AST
13     .use(rehypeSanitize) // Sanitize HTML input
14     .use(rehypeStringify) // Convert AST into serialized HTML
15     .process('Hello, Next.js!');
16
17   console.log(String(file)); // <p>Hello, Next.js!</p>
18 }
```

The `remark` and `rehype` ecosystem contains plugins for [syntax highlighting ↗](#), [linking headings ↗](#), [generating a table of contents ↗](#), and more.

When using `@next/mdx` as shown below, you **do not** need to use `remark` or `rehype` directly, as it is handled for you.

Using the Rust-based MDX compiler (Experimental)

Next.js supports a new MDX compiler written in Rust. This compiler is still experimental and is not recommended for production use. To use the new compiler, you need to configure `next.config.js` when you pass it to `withMDX`:

 next.config.js



```
1 module.exports = withMDX({
2   experimental: {
3     mdxRs: true,
4   },
5 });
```

Helpful Links

- [MDX ↗](#)
- [@next/mdx ↗](#)
- [remark ↗](#)
- [rehype ↗](#)

> Menu

Pages Router > ... > Configuring > PostCSS

PostCSS

▼ Examples

- [Tailwind CSS Example ↗](#)
-

Default Behavior

Next.js compiles CSS for its [built-in CSS support](#) using PostCSS.

Out of the box, with no configuration, Next.js compiles CSS with the following transformations:

- [Autoprefixer ↗](#) automatically adds vendor prefixes to CSS rules (back to IE11).
- [Cross-browser Flexbox bugs ↗](#) are corrected to behave like [the spec ↗](#).
- New CSS features are automatically compiled for Internet Explorer 11 compatibility:
 - [all Property ↗](#)
 - [Break Properties ↗](#)
 - [font-variant Property ↗](#)
 - [Gap Properties ↗](#)
 - [Media Query Ranges ↗](#)

By default, [CSS Grid ↗](#) and [Custom Properties ↗](#) (CSS variables) are **not compiled** for IE11 support.

To compile [CSS Grid Layout ↗](#) for IE11, you can place the following comment at the top of your CSS file:

```
/* autoprefixer grid: autoplace */
```

You can also enable IE11 support for [CSS Grid Layout](#) in your entire project by configuring autoprefixer with the configuration shown below (collapsed). See "[Customizing Plugins](#)" below for more information.

► [Click to view the configuration to enable CSS Grid Layout](#)

CSS variables are not compiled because it is [not possible to safely do so](#). If you must use variables, consider using something like [Sass variables](#) which are compiled away by [Sass](#).

Customizing Target Browsers

Next.js allows you to configure the target browsers (for [Autoprefixer](#) and compiled css features) through [Browserslist](#).

To customize browserslist, create a `browserslist` key in your `package.json` like so:

```
1  {
2    "browserslist": [ ">0.3%", "not dead", "not op_mini all" ]
3 }
```

You can use the [browserslist](#) tool to visualize what browsers you are targeting.

CSS Modules

No configuration is needed to support CSS Modules. To enable CSS Modules for a file, rename the file to have the extension `.module.css`.

You can learn more about [Next.js' CSS Module support here](#).

Customizing Plugins

Warning: When you define a custom PostCSS configuration file, Next.js **completely disables** the [default behavior](#). Be sure to manually configure all the features you need compiled, including [Autoprefixer](#). You also need to install any plugins included in your custom configuration manually, i.e.

```
npm install postcss-flexbugs-fixes postcss-preset-env
```

To customize the PostCSS configuration, create a `postcss.config.json` file in the root of your project.

This is the default configuration used by Next.js:

 postcss.config.json

```
1  {
2    "plugins": [
3      "postcss-flexbugs-fixes",
4      [
5        "postcss-preset-env",
6        {
7          "autoprefixer": {
8            "flexbox": "no-2009"
9          },
10         "stage": 3,
11         "features": {
12           "custom-properties": false
13         }
14       }
15     ]
16   ]
17 }
```

Note: Next.js also allows the file to be named `.postcssrc.json`, or, to be read from the `postcss` key in `package.json`.

It is also possible to configure PostCSS with a `postcss.config.js` file, which is useful when you want to conditionally include plugins based on environment:

 postcss.config.js

```
1 module.exports = {
2   plugins:
3     process.env.NODE_ENV === 'production'
4     ? [
5       'postcss-flexbugs-fixes',
6       [
7         'postcss-preset-env',
8         {
9           autoprefixer: {
10             flexbox: 'no-2009',
11           },
12           stage: 3,
13           features: {
14             'custom-properties': false,
15           },
16           },
17         ],
18       ],
19     ]}
```

```
18 ]
19   : [
20     // No transformations in development
21   ],
22 };
```

Note: Next.js also allows the file to be named `.postcssrc.js`.

Do **not use `require()`** to import the PostCSS Plugins. Plugins must be provided as strings.

Note: If your `postcss.config.js` needs to support other non-Next.js tools in the same project, you must use the interoperable object-based format instead:

```
1 module.exports = {
2   plugins: {
3     'postcss-flexbugs-fixes': {},
4     'postcss-preset-env': {
5       autoprefixer: {
6         flexbox: 'no-2009',
7       },
8       stage: 3,
9       features: {
10         'custom-properties': false,
11       },
12     },
13   },
14 };
```

> Menu

Pages Router > ... > Configuring > Preview Mode (Legacy)

Preview Mode (Legacy)

Warning: This feature is **legacy** and is superseded by [Draft Mode](#).

► Examples

In the [Pages documentation](#) and the [Data Fetching documentation](#), we talked about how to pre-render a page at build time (**Static Generation**) using `getStaticProps` and `getStaticPaths`.

Static Generation is useful when your pages fetch data from a headless CMS. However, it's not ideal when you're writing a draft on your headless CMS and want to **preview** the draft immediately on your page. You'd want Next.js to render these pages at **request time** instead of build time and fetch the draft content instead of the published content. You'd want Next.js to bypass Static Generation only for this specific case.

Next.js has a feature called **Preview Mode** which solves this problem. Here are instructions on how to use it.

Step 1: Create and access a preview API route

Take a look at the [API Routes documentation](#) first if you're not familiar with Next.js API Routes.

First, create a **preview API route**. It can have any name - e.g. `pages/api/preview.js` (or `.ts` if using TypeScript).

In this API route, you need to call `setPreviewData` on the response object. The argument for `setPreviewData` should be an object, and this can be used by `getStaticProps` (more on this later). For now, we'll use `{}`.

```
1 export default function handler(req, res) {  
2   // ...  
3   res.setPreviewData({});  
4   // ...
```

`res.setPreviewData` sets some **cookies** on the browser which turns on the preview mode. Any requests to Next.js containing these cookies will be considered as the **preview mode**, and the behavior for statically generated pages will change (more on this later).

You can test this manually by creating an API route like below and accessing it from your browser manually:

JS pages/api/preview.js

```
1 // simple example for testing it manually from your browser.
2 export default function handler(req, res) {
3   res.setPreviewData({});
4   res.end('Preview mode enabled');
5 }
```

If you open your browser's developer tools and visit `/api/preview`, you'll notice that the `__prerender_bypass` and `__next_preview_data` cookies will be set on this request.

Securely accessing it from your Headless CMS

In practice, you'd want to call this API route *securely* from your headless CMS. The specific steps will vary depending on which headless CMS you're using, but here are some common steps you could take.

These steps assume that the headless CMS you're using supports setting **custom preview URLs**. If it doesn't, you can still use this method to secure your preview URLs, but you'll need to construct and access the preview URL manually.

First, you should create a **secret token string** using a token generator of your choice. This secret will only be known by your Next.js app and your headless CMS. This secret prevents people who don't have access to your CMS from accessing preview URLs.

Second, if your headless CMS supports setting custom preview URLs, specify the following as the preview URL. This assumes that your preview API route is located at `pages/api/preview.js`.

>_ Terminal

```
https://<your-site>/api/preview?secret=<token>&slug=<path>
```

- `<your-site>` should be your deployment domain.
- `<token>` should be replaced with the secret token you generated.

- <path> should be the path for the page that you want to preview. If you want to preview /posts/foo , then you should use &slug=/posts/foo .

Your headless CMS might allow you to include a variable in the preview URL so that <path> can be set dynamically based on the CMS's data like so: &slug=/posts/{entry.fields.slug}

Finally, in the preview API route:

- Check that the secret matches and that the slug parameter exists (if not, the request should fail).
- Call res.setPreviewData .
- Then redirect the browser to the path specified by slug . (The following example uses a 307 redirect ↗).

```
1 export default async (req, res) => {
2   // Check the secret and next parameters
3   // This secret should only be known to this API route and the CMS
4   if (req.query.secret !== 'MY_SECRET_TOKEN' || !req.query.slug) {
5     return res.status(401).json({ message: 'Invalid token' });
6   }
7
8   // Fetch the headless CMS to check if the provided `slug` exists
9   // getPostBySlug would implement the required fetching logic to the headless CMS
10  const post = await getPostBySlug(req.query.slug);
11
12  // If the slug doesn't exist prevent preview mode from being enabled
13  if (!post) {
14    return res.status(401).json({ message: 'Invalid slug' });
15  }
16
17  // Enable Preview Mode by setting the cookies
18  res.setPreviewData({});
19
20  // Redirect to the path from the fetched post
21  // We don't redirect to req.query.slug as that might lead to open redirect vulnerability
22  res.redirect(post.slug);
23};
```

If it succeeds, then the browser will be redirected to the path you want to preview with the preview mode cookies being set.

Step 2: Update getStaticProps

The next step is to update getStaticProps to support the preview mode.

If you request a page which has `getStaticProps` with the preview mode cookies set (via `res.setPreviewData`), then `getStaticProps` will be called at **request time** (instead of at build time).

Furthermore, it will be called with a `context` object where:

- `context.preview` will be `true`.
- `context.previewData` will be the same as the argument used for `setPreviewData`.

```
1 export async function getStaticProps(context) {  
2   // If you request this page with the preview mode cookies set:  
3   //  
4   // - context.preview will be true  
5   // - context.previewData will be the same as  
6   //   the argument used for `setPreviewData`.  
7 }
```

We used `res.setPreviewData({})` in the preview API route, so `context.previewData` will be `{}`. You can use this to pass session information from the preview API route to `getStaticProps` if necessary.

If you're also using `getStaticPaths`, then `context.params` will also be available.

Fetch preview data

You can update `getStaticProps` to fetch different data based on `context.preview` and/or `context.previewData`.

For example, your headless CMS might have a different API endpoint for draft posts. If so, you can use `context.preview` to modify the API endpoint URL like below:

```
1 export async function getStaticProps(context) {  
2   // If context.preview is true, append "/preview" to the API endpoint  
3   // to request draft data instead of published data. This will vary  
4   // based on which headless CMS you're using.  
5   const res = await fetch(`https://.../${context.preview ? 'preview' : ''}`);  
6   // ...  
7 }
```

That's it! If you access the preview API route (with `secret` and `slug`) from your headless CMS or manually, you should now be able to see the preview content. And if you update your draft without publishing, you should be able to preview the draft.

Set this as the preview URL on your headless CMS or access manually, and you should be able to see the preview.



```
https://<your-site>/api/preview?secret=<token>&slug=<path>
```



More Details

Note: during rendering `next/router` exposes an `isPreview` flag, see the [router object docs](#) for more info.



Specify the Preview Mode duration

`setPreviewData` takes an optional second parameter which should be an options object. It accepts the following keys:

- `maxAge` : Specifies the number (in seconds) for the preview session to last for.
- `path` : Specifies the path the cookie should be applied under. Defaults to `/` enabling preview mode for all paths.

```
1 setPreviewData(data, {  
2   maxAge: 60 * 60, // The preview mode cookies expire in 1 hour  
3   path: '/about', // The preview mode cookies apply to paths with /about  
4 });
```



Clear the Preview Mode cookies

By default, no expiration date is set for Preview Mode cookies, so the preview session ends when the browser is closed.

To clear the Preview Mode cookies manually, create an API route that calls `clearPreviewData()`:

js pages/api/clear-preview-mode-cookies.js



```
1 export default function handler(req, res) {  
2   res.clearPreviewData({});  
3 }
```



Then, send a request to `/api/clear-preview-mode-cookies` to invoke the API Route. If calling this route using `next/link`, you must pass `prefetch={false}` to prevent calling `clearPreviewData` during link



prefetching.

If a path was specified in the `setPreviewData` call, you must pass the same path to `clearPreviewData`:

JS pages/api/clear-preview-mode-cookies.js

```
1 export default function handler(req, res) {
2   const { path } = req.query;
3
4   res.clearPreviewData({ path });
5 }
```

previewData size limits

You can pass an object to `setPreviewData` and have it be available in `getStaticProps`. However, because the data will be stored in a cookie, there's a size limitation. Currently, preview data is limited to 2KB.

Works with getServerSideProps

The preview mode works on `getServerSideProps` as well. It will also be available on the `context` object containing `preview` and `previewData`.

Works with API Routes

API Routes will have access to `preview` and `previewData` under the request object. For example:

```
1 export default function myApiRoute(req, res) {
2   const isPreview = req.preview;
3   const previewData = req.previewData;
4   // ...
5 }
```

Unique per next build

Both the bypass cookie value and the private key for encrypting the `previewData` change when `next build` is completed. This ensures that the bypass cookie can't be guessed.

Note: To test Preview Mode locally over HTTP your browser will need to allow third-party cookies and local storage access.

> Menu

Pages Router > ... > Configuring > src Directory

src Directory

As an alternative to having the special Next.js `app` or `pages` directories in the root of your project, Next.js also supports the common pattern of placing application code under the `src` directory.

This separates application code from project configuration files which mostly live in the root of a project. Which is preferred by some individuals and teams.

To use the `src` directory, move the `app` Router folder or `pages` Router folder to `src/app` or `src/pages` respectively.

Good to know

- The `/public` directory should remain in the root of your project.
- Config files like `package.json`, `next.config.js` and `tsconfig.json` should remain in the root of your project.
- `.env.*` files should remain in the root of your project.
- `src/app` or `src/pages` will be ignored if `app` or `pages` are present in the root directory.
- If you're using `src`, you'll probably also move other application folders such as `/components` or `/lib`.

> Menu

Pages Router > ... > Configuring > TypeScript

TypeScript

Next.js provides a TypeScript-first development experience for building your React application.

It comes with built-in TypeScript support for automatically installing the necessary packages and configuring the proper settings.

New Projects

`create-next-app` now ships with TypeScript by default.

> Terminal



```
npx create-next-app@latest
```

Existing Projects

Add TypeScript to your project by renaming a file to `.ts` / `.tsx`. Run `next dev` and `next build` to automatically install the necessary dependencies and add a `tsconfig.json` file with the recommended config options.

Minimum TypeScript Version

It is highly recommended to be on at least `v4.5.2` of TypeScript to get syntax features such as [type modifiers on import names](#) [↗] and [performance improvements](#) [↗].

Static Generation and Server-side Rendering

For `getStaticProps`, `getStaticPaths`, and `getServerSideProps`, you can use the `GetStaticProps`, `GetStaticPaths`, and `GetServerSideProps` types respectively:

TS pages/blog/[slug].tsx

```
1 import { GetStaticProps, GetStaticPaths, GetServerSideProps } from 'next';
2
3 export const getStaticProps: GetStaticProps = async (context) => {
4   // ...
5 };
6
7 export const getStaticPaths: GetStaticPaths = async () => {
8   // ...
9 };
10
11 export const getServerSideProps: GetServerSideProps = async (context) => {
12   // ...
13 };
```

API Routes

The following is an example of how to use the built-in types for API routes:

```
1 import type { NextApiRequest, NextApiResponse } from 'next';
2
3 export default function handler(req: NextApiRequest, res: NextApiResponse) {
4   res.status(200).json({ name: 'John Doe' });
5 }
```

You can also type the response data:

```
1 import type { NextApiRequest, NextApiResponse } from 'next';
2
3 type Data = {
4   name: string;
5 };
6
7 export default function handler(
8   req: NextApiRequest,
9   res: NextApiResponse<Data>,
```

```
10 ) {
11   res.status(200).json({ name: 'John Doe' });
12 }
```

Custom App

If you have a [custom App](#), you can use the built-in type [AppProps](#) and change file name to

```
./pages/_app.tsx
```

 like so:

```
1 import type { AppProps } from 'next/app';
2
3 export default function MyApp({ Component, pageProps }: AppProps) {
4   return <Component {...pageProps} />;
5 }
```

Path aliases and baseUrl

Next.js automatically supports the [tsconfig.json](#) "paths" and "baseUrl" options.

You can learn more about this feature on the [Module Path aliases documentation](#).

Type checking next.config.js

The [next.config.js](#) file must be a JavaScript file as it does not get parsed by Babel or TypeScript, however you can add some type checking in your IDE using JSDoc as below:

```
1 // @ts-check
2
3 /**
4  * @type {import('next').NextConfig}
5 */
6 const nextConfig = {
7   /* config options here */
8 };
9
10 module.exports = nextConfig;
```

Incremental type checking

Since v10.2.1 Next.js supports [incremental type checking](#) ↗ when enabled in your `tsconfig.json`, this can help speed up type checking in larger applications.

Ignoring TypeScript Errors

Next.js fails your **production build** (`next build`) when TypeScript errors are present in your project.

If you'd like Next.js to dangerously produce production code even when your application has errors, you can disable the built-in type checking step.

If disabled, be sure you are running type checks as part of your build or deploy process, otherwise this can be very dangerous.

Open `next.config.js` and enable the `ignoreBuildErrors` option in the `typescript` config:

js next.config.js

```
1 module.exports = {
2   typescript: {
3     // !!! WARN !!!
4     // Dangerously allow production builds to successfully complete even if
5     // your project has type errors.
6     // !!! WARN !!!
7     ignoreBuildErrors: true,
8   },
9 }
```

Version Changes

Version	Changes
---------	---------

v13.2.0	Statically typed links are available in beta
---------	--

v12.0.0	SWC is now used by default to compile TypeScript and TSX for faster builds.
---------	---

v10.2.1	Incremental type checking ↗ support added when enabled in your <code>tsconfig.json</code> .
---------	---

> Menu

Pages Router > Building Your Application > Data Fetching

Data Fetching

Data fetching in Next.js allows you to render your content in different ways, depending on your application's use case. These include pre-rendering with **Server-side Rendering** or **Static Generation**, and updating or creating content at runtime with **Incremental Static Regeneration**.

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 ↗ \(Demo ↗\)](#)
- [Prepr Example ↗ \(Demo ↗\)](#)
- [Agility CMS Example ↗ \(Demo ↗\)](#)
- [Cosmic Example ↗ \(Demo ↗\)](#)
- [ButterCMS Example ↗ \(Demo ↗\)](#)
- [Storyblok Example ↗ \(Demo ↗\)](#)
- [GraphCMS Example ↗ \(Demo ↗\)](#)
- [Kontent Example ↗ \(Demo ↗\)](#)
- [Static Tweet Demo ↗](#)
- [Enterspeed Example ↗ \(Demo ↗\)](#)

[getStaticProps](#)

Fetch data and generate static pages with `getStaticProps`. Learn more about this API for data fetching in Next.js.

[getStaticPaths](#)

Fetch data and generate static pages with `getStaticProps`. Learn more about this API for data fetching in Next.js.

[getServerSideProps](#)

Fetch data on each request with `getServerSideProps`.

[Incremental Static Regeneration](#)

Learn how to create or update static pages at runtime with Incremental Static Regeneration.

[Client-side Fetching](#)

Learn about client-side data fetching, and how to use SWR, a data fetching React hook library that handles caching, revalidation, focus tracking, refetching on interval and more.

[Building Forms](#)

Learn how to create forms with Next.js, from the form HTML element to advanced concepts with React.

Building Forms

A web form has a **client-server** relationship. They are used to send data handled by a web server for processing and storage. The form itself is the client, and the server is any storage mechanism that can be used to store, retrieve and send data when needed.

This guide will teach you how to create a web form with Next.js.

Part 1: HTML Form

HTML forms are built using the `<form>` tag. It takes a set of attributes and fields to structure the form for features like text fields, checkboxes, dropdown menus, buttons, radio buttons, etc.

Here's the syntax of an HTML form:

```
1 <!-- Basic HTML Form -->
2 <form action="/send-data-here" method="post">
3   <label for="first">First name:</label>
4   <input type="text" id="first" name="first" />
5   <label for="last">Last name:</label>
6   <input type="text" id="last" name="last" />
7   <button type="submit">Submit</button>
8 </form>
```

The front-end looks like this:

First Name

Last Name

Submit

The HTML `<form>` tag acts as a container for different `<input>` elements like `text` field and submit button. Let's study each of these elements:

- `action` : An attribute that specifies where the form data is sent when the form is submitted. It's generally a URL (an absolute URL or a relative URL).
- `method` : Specifies the [HTTP method](#), i.e., `GET` or `POST` used to send data while submitting the form.
- `<label>` : An element that defines the label for other form elements. Labels aid accessibility, especially for screen readers.
- `<input>` : The form element that is widely used to structure the form fields. It depends significantly on the value of the `type` attribute. Input types can be `text`, `checkbox`, `email`, `radio`, and more.
- `<button>` : Represents a clickable button that's used to submit the form data.

Form Validation

A process that checks if the information provided by a user is correct or not. Form validation also ensures that the provided information is in the correct format (e.g. there's an @ in the email field). These are of two types:

- **Client-side:** Validation is done in the browser
- **Server-side:** Validation is done on the server

Though both of these types are equally important, this guide will focus on client-side validation only.

Client-side validation is further categorized as:

- **Built-in:** Uses HTML-based attributes like `required`, `type`, `minLength`, `maxLength`, `pattern`, etc.
- **JavaScript-based:** Validation that's coded with JavaScript.

Built-in Form Validation Using `required`, `type`, `minLength`, `maxLength`

- `required`: Specifies which fields must be filled before submitting the form.
- `type`: Specifies the data's type (i.e a number, email address, string, etc).
- `minLength`: Specifies minimum length for the text data string.
- `maxLength`: Specifies maximum length for the text data string.

So, a form using this attributes may look like:

```
1 <!-- HTML Form with Built-in Validation -->
2 <form action="/send-data-here" method="post">
3   <label for="roll">Roll Number</label>
4   <input
5     type="text"
6     id="roll"
7     name="roll"
8     required
9     minlength="10"
10    maxlength="20"
11  />
12  <label for="name">Name:</label>
13  <input type="text" id="name" name="name" required />
14  <button type="submit">Submit</button>
15 </form>
```

With these validation checks in place, when a user tries to submit an empty field for Name, it gives an error that pops right in the form field. Similarly, a roll number can only be entered if it's 10-20 characters long.

Roll Number

123

Name



Please lengthen this text to 10 characters or more (you are currently using 3 characters).

Submit

JavaScript-based Form Validation

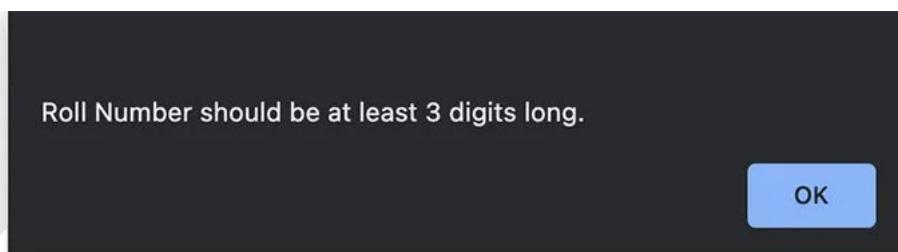
Form Validation is important to ensure that a user has submitted the correct data, in a correct format. JavaScript offers an additional level of validation along with HTML native form attributes on the client side. Developers generally prefer validating form data through JavaScript because its data processing is faster when compared to server-side validation, however front-end validation may be less secure in some scenarios as a malicious user could always send malformed data to your server.

The following example shows using JavaScript to validate a form:

```
1 <form onsubmit="validateFormWithJS()">
2   <label for="rollNumber">Roll Number:</label>
3   <input type="text" name="rollNumber" id="rollNumber" />
4
5   <label for="name">Name:</label>
6   <input type="text" name="name" id="name" />
7
8   <button type="submit">Submit</button>
9 </form>
10
11 <script>
12   function validateFormWithJS() {
13     const name = document.querySelector('#name').value;
```

```
14     const rollNumber = document.querySelector('#rollNumber').value;
15
16     if (!name) {
17         alert('Please enter your name.');
18         return false;
19     }
20
21     if (rollNumber.length < 3) {
22         alert('Roll Number should be at least 3 digits long.');
23         return false;
24     }
25 }
26 </script>
```

The HTML `script` tag is used to embed any client-side JavaScript. It can either contain inline scripting statements (as shown in the example above) or point to an external script file via the `src` attribute. This example validates the name and roll number of a user. The `validateFormWithJS()` function does not allow an empty name field, and the roll number must be at least three digits long. The validation is performed when you hit the Submit button. You are not redirected to the next page until the given values are correct.



Name:

Roll Number:

JavaScript validation with Regular Expressions uses the `pattern` HTML attribute. A regular expression (commonly known as RegEx) is an object that describes a pattern of characters. You can only apply the `pattern` attribute to the `<input>` element. This way, you can validate the input value using Regular Expressions (RegEx) by defining your own rules. Once again, if the value does not match the defined pattern, the input will give an error. The below example shows using the `pattern` attribute on an `input` element:

```
1 <form action="/action_page.php">
2   <label for="pswrd">Password:</label>
3   <input
4     type="password"
5     id="pswrd"
6     name="pswrd"
7     pattern="[a-zA-Z0-9]{1,15}"
8     title="Password should be digits (0 to 9) or alphabets (a to z)."
9   />
10
11   <button type="submit">Submit</button>
12 </form>
```

The password form field must only contain digits (0 to 9), lowercase alphabets (a to z) and it must be no more than 15 characters in length. No other characters (#,\$,&, etc.) are allowed. The rule in RegEx is written as `[a-zA-Z0-9]{1,15}`.

Password: **helloworld123#%**

 Please match the format requested.
Password should be digits (0 to 9) or
alphabets (a to z)

Part 2: Project Setup

In the following section you will be creating forms in React using Next.js.

Create a new Next.js app. You can use the [create-next-app](#) for a quick start. In your command line terminal, run the following:

```
npx create-next-app
```

Answer the questions to create your project, and give it a name, this example uses [next-forms](#). Next `cd` into this directory, and run `npm run dev` or `yarn dev` command to start the development server.

Open the URL printed in the terminal to ensure that your app is running successfully.

Part 3: Setting up a Next.js Form API Route

Both the client and the server will be built using Next.js. For the server part, create an API endpoint where you will send the form data.

Next.js offers a file-based system for routing that's built on the [concept of pages](#). Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a page. This [API endpoint](#) is going to be server-side only.

Go to `pages/api`, create a file called `form.js` and paste this code written in Node.js:

```
1 export default function handler(req, res) {
2   // Get data submitted in request's body.
3   const body = req.body;
4
5   // Optional logging to see the responses
6   // in the command line where next.js app is running.
7   console.log('body: ', body);
8
9   // Guard clause checks for first and last name,
10  // and returns early if they are not found
11  if (!body.first || !body.last) {
```

```
12 // Sends a HTTP bad request error code
13     return res.status(400).json({ data: 'First or last name not found' });
14 }
15
16 // Found the name.
17 // Sends a HTTP success code
18 res.status(200).json({ data: `${body.first} ${body.last}` });
19 }
```

This form `handler` function will receive the request `req` from the client (i.e. submitted form data). And in return, it'll send a response `res` as JSON that will have both the first and the last name. You can access this API endpoint at `http://localhost:3000/api/form` or replace the localhost URL with an actual Vercel deployment when you deploy.

Moreover, you can also attach this API to a database like MongoDB or Google Sheets. This way, your submitted form data will be securely stored for later use. For this guide, no database is used. Instead, the same data is returned to the user to demo how it's done.

Form Submission without JavaScript

You can now use `/api/form` relative endpoint inside the `action` attribute of the form. You are sending form data to the server when the form is submitted via `POST` HTTP method (which is used to send data).

```
1 <form action="/api/form" method="post">
2   <label for="first">First name:</label>
3   <input type="text" id="first" name="first" />
4   <label for="last">Last name:</label>
5   <input type="text" id="last" name="last" />
6   <button type="submit">Submit</button>
7 </form>
```

If you submit this form, it will submit the data to the forms API endpoint `/api/form`. The server then responds, generally handling the data and loading the URL defined by the `action` attribute, causing a new page load. So in this case you'll be redirected to `http://localhost:3000/api/form` with the following response from the server.

```
{
  "data": "Maedah Batool"
}
```

Part 4: Configuring Forms in Next.js

You have created a Next.js API Route for form submission. Now it's time to configure the client (the form itself) inside Next.js using React. The first step will be extending your knowledge of HTML forms and converting it into React (using [JSX ↗](#)).

Here's the same form in a [React function component ↗](#) written using [JSX ↗](#).

```

1  export default function Form() {
2    return (
3      <form action="/api/form" method="post">
4        <label htmlFor="first">First Name</label>
5        <input type="text" id="first" name="first" required />
6
7        <label htmlFor="last">Last Name</label>
8        <input type="text" id="last" name="last" required />
9
10       <button type="submit">Submit</button>
11     </form>
12   );
13 }
```

Here's what changed:

- The `for` attribute is changed to `htmlFor`. (Since `for` is a keyword associated with the "for" loop in JavaScript, React elements use `htmlFor` instead.)
- The `action` attribute now has a relative URL which is the form API endpoint.

This completes the basic structure of your Next.js-based form.

You can view the entire source code of [next-forms ↗](#) example repo that we're creating here as a working example. Feel free to clone it and start right away. This demo is built with `create-next-app`, and you can preview the basic form CSS

Forms with Next.js!

Get started by looking at `pages/js-form.js` and `pages/no-js-form.js`

Form with JavaScript →

Learn to handle forms with JavaScript in Next.js.

Form without JavaScript →

Learn to handle forms without JavaScript in Next.js.

Part 5: Form Submission without JavaScript

JavaScript brings interactivity to our web applications, but sometimes you need to control the JavaScript bundle from being too large, or your site's visitors might have JavaScript disabled.

There are several reasons why users disable JavaScript:

- Addressing bandwidth constraints
- Increasing device (phone or laptop) battery life
- For privacy so they won't be tracked with analytical scripts

Regardless of the reason, disabling JavaScript will impact site functionality partially, if not completely.

Next open the `next-forms` directory. Inside the `/pages` directory, create a file `no-js-form.js`.

Quick Tip: In Next.js, a page is a React Component exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the Pages Router. Each page is associated with a route based on its file name.

Example: If you create `pages/no-js-form.js`, it will be accessible at `your-domain.tld/no-js-form`.

Let's use the same code from above:

```

1 export default function PageWithoutJSbasedForm() {
2   return (
3     <form action="/api/form" method="post">
4       <label htmlFor="first">First Name</label>
5       <input type="text" id="first" name="first" required />
6
7       <label htmlFor="last">Last Name</label>
8       <input type="text" id="last" name="last" required />
9
10      <button type="submit">Submit</button>
11    </form>
12  );
13}

```

With JavaScript disabled, when you hit the Submit button, an event is triggered, which collects the form data and sends it to our forms API endpoint as defined in the `action` attribute and using `POST` HTTP `method`. You'll be redirected to the `/api/form` endpoint since that's how form `action` works.

The form data will be submitted on the server as a request `req` to the form handler function written above. It will process the data and return a JSON string as a response `res` with your submitted name included.

To improve the experience here, as a response you can redirect the user to a page and thank them for submitting the form.

Part 6: Form Submission with JavaScript Enabled

Inside `/pages`, you'll create another file called `js-form.js`. This will create a `/js-form` page on your Next.js app.

Now, as soon as the form is submitted, we prevent the form's default behavior of reloading the page. We'll take the form data, convert it to JSON string, and send it to our server, the API endpoint. Finally, our server will respond with the name submitted. All of this with a basic JavaScript `handleSubmit()` function.

Here's what this function looks like. It's well documented for you to understand each step:

```

1 export default function PageWithJSbasedForm() {
2   // Handles the submit event on form submit.
3   const handleSubmit = async (event) => {
4     // Stop the form from submitting and refreshing the page.
5     event.preventDefault();
6
7     // Get data from the form.
8     const data = {
9       first: event.target.first.value,
10      last: event.target.last.value
11    };
12
13    // Convert the data to a JSON object.
14    const jsonData = JSON.stringify(data);
15
16    // Create a new FormData object.
17    const formData = new FormData();
18
19    // Add the JSON data to the form data.
20    formData.append("data", jsonData);
21
22    // Set the form data to the FormData object.
23    event.target.setData(formData);
24
25    // Submit the form.
26    event.target.submit();
27  };
28}

```

```

9      first: event.target.first.value,
10     last: event.target.last.value,
11   };
12
13   // Send the data to the server in JSON format.
14   const JSONdata = JSON.stringify(data);
15
16   // API endpoint where we send form data.
17   const endpoint = '/api/form';
18
19   // Form the request for sending data to the server.
20   const options = {
21     // The method is POST because we are sending data.
22     method: 'POST',
23     // Tell the server we're sending JSON.
24     headers: {
25       'Content-Type': 'application/json',
26     },
27     // Body of the request is the JSON data we created above.
28     body: JSONdata,
29   };
30
31   // Send the form data to our forms API on Vercel and get a response.
32   const response = await fetch(endpoint, options);
33
34   // Get the response data from server as JSON.
35   // If server returns the name submitted, that means the form works.
36   const result = await response.json();
37   alert(`Is this your full name: ${result.data}`);
38 };
39 return (
40   // We pass the event to the handleSubmit() function on submit.
41   <form onSubmit={handleSubmit}>
42     <label htmlFor="first">First Name</label>
43     <input type="text" id="first" name="first" required />
44
45     <label htmlFor="last">Last Name</label>
46     <input type="text" id="last" name="last" required />
47
48     <button type="submit">Submit</button>
49   </form>
50 );
51 }

```

It's a Next.js page with a React function component called `PageWithJSbasedForm` with a `<form>` element written in JSX. There's no action on the `<form>` element. Instead, we use the `onSubmit` event handler to send data to our `{handleSubmit}` function.

The `handleSubmit()` function processes your form data through a series of steps:

- The `event.preventDefault()` stops the `<form>` element from refreshing the entire page.
- We created a JavaScript object called `data` with the `first` and `last` values from the form.

- JSON is a language-agnostic data transfer format. So we use `JSON.stringify(data)` to convert the data to JSON.
 - We then use `fetch()` to send the data to our `/api/form` endpoint using JSON and HTTP `POST` method.
 - Server sends back a response with the name submitted. Woohoo!
-

Conclusion

This guide has covered the following:

- The basic HTML `form` element
- Understanding forms with React.js
- Validating forms data with and without JavaScript
- Using Next.js API Routes to handle `req` and `res` from the client and server

For more details go through [Next.js Learn Course ↗](#).

> Menu

Pages Router > ... > Data Fetching > Client-side Fetching

Client-side Fetching

Client-side data fetching is useful when your page doesn't require SEO indexing, when you don't need to pre-render your data, or when the content of your pages needs to update frequently. Unlike the server-side rendering APIs, you can use client-side data fetching at the component level.

If done at the page level, the data is fetched at runtime, and the content of the page is updated as the data changes. When used at the component level, the data is fetched at the time of the component mount, and the content of the component is updated as the data changes.

It's important to note that using client-side data fetching can affect the performance of your application and the load speed of your pages. This is because the data fetching is done at the time of the component or pages mount, and the data is not cached.

Client-side data fetching with useEffect

The following example shows how you can fetch data on the client side using the useEffect hook.

```
1 import { useState, useEffect } from 'react';
2
3 function Profile() {
4   const [data, setData] = useState(null);
5   const [isLoading, setLoading] = useState(false);
6
7   useEffect(() => {
8     setLoading(true);
9     fetch('/api/profile-data')
10       .then((res) => res.json())
11       .then((data) => {
12         setData(data);
13         setLoading(false);
14       });
15   }, []);
16
17   if (isLoading) return <p>Loading...</p>;
```

```
18 if (!data) return <p>No profile data</p>;
19
20 return (
21   <div>
22     <h1>{data.name}</h1>
23     <p>{data.bio}</p>
24   </div>
25 );
26 }
```

Client-side data fetching with SWR

The team behind Next.js has created a React hook library for data fetching called [SWR ↗](#). It is **highly recommended** if you are fetching data on the client-side. It handles caching, revalidation, focus tracking, refetching on intervals, and more.

Using the same example as above, we can now use SWR to fetch the profile data. SWR will automatically cache the data for us and will revalidate the data if it becomes stale.

For more information on using SWR, check out the [SWR docs ↗](#).

```
1 import useSWR from 'swr';
2
3 const fetcher = (...args) => fetch(...args).then((res) => res.json());
4
5 function Profile() {
6   const { data, error } = useSWR('/api/profile-data', fetcher);
7
8   if (error) return <div>Failed to load</div>;
9   if (!data) return <div>Loading...</div>;
10
11  return (
12    <div>
13      <h1>{data.name}</h1>
14      <p>{data.bio}</p>
15    </div>
16  );
17 }
```

> Menu

Pages Router > ... > Data Fetching > getServerSideProps

getServerSideProps

If you export a function called `getServerSideProps` (Server-Side Rendering) from a page, Next.js will pre-render this page on each request using the data returned by `getServerSideProps`.

TS pages/index.tsx

```
1 import type { InferGetServerSidePropsType, GetServerSideProps } from 'next';
2
3 type Repo = {
4   name: string;
5   stargazers_count: number;
6 };
7
8 export const getServerSideProps: GetServerSideProps<{
9   repo: Repo;
10 }> = async () => {
11   const res = await fetch('https://api.github.com/repos/vercel/next.js');
12   const repo = await res.json();
13   return { props: { repo } };
14 };
15
16 export default function Page({
17   repo,
18 }: InferGetServerSidePropsType<typeof getServerSideProps>) {
19   return repo.stargazers_count;
20 }
```

Note that irrespective of rendering type, any `props` will be passed to the page component and can be viewed on the client-side in the initial HTML. This is to allow the page to be [hydrated](#) correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

When does getServerSideProps run

`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 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` or `next/router`, Next.js sends an API request to the server, which runs `getServerSideProps`

`getServerSideProps` returns JSON which 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 the [next-code-elimination tool](#) ↗ to verify what Next.js eliminates from the client-side bundle.

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

Note that you must export `getServerSideProps` as a standalone function — it will **not** work if you add `getServerSideProps` as a property of the page component.

The [getServerSideProps API reference](#) covers all parameters and props that can be used with `getServerSideProps`.

When should I use `getServerSideProps`

You should use `getServerSideProps` only if you need to render a page whose data must be fetched at request time. This could be due to the nature of the data or properties of the request (such as `authorization` headers or geo location). Pages using `getServerSideProps` will be server side rendered at request time and only be cached if [cache-control headers are configured](#).

If you do not need to render the data during the request, then you should consider fetching data on the [client side](#) or `getStaticProps`.

`getServerSideProps` or API Routes

It can be tempting to reach for an [API Route](#) when you want to fetch data from the server, then call that API route from `getServerSideProps`. This is an unnecessary and inefficient approach, as it will cause an extra request to be made due to both `getServerSideProps` and API Routes running on the server.

Take the following example. An API route is used to fetch some data from a CMS. That API route is then called directly from `getServerSideProps`. This produces an additional call, reducing performance. Instead,

directly import the logic used inside your API Route into `getServerSideProps`. This could mean calling a CMS, database, or other API directly from inside `getServerSideProps`.

getServerSideProps with Edge API Routes

`getServerSideProps` can be used with both [Serverless](#) and [Edge Runtimes](#), and you can set props in both. However, currently in the Edge Runtime, you do not have access to the response object. This means that you cannot — for example — add cookies in `getServerSideProps`. To have access to the response object, you should **continue to use the Node.js runtime**, which is the default runtime.

You can explicitly set the runtime on a per-page basis by modifying the `config`, for example:

pages/index.tsx

```
1 export const config = {  
2   runtime: 'nodejs', // or "edge"  
3 };  
4  
5 export const getServerSideProps = async () => {};
```



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:

- 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.

Using getServerSideProps to fetch data at request time

The following example shows how to fetch data at request time and pre-render the result.

```
1 function Page({ data }) {
2   // Render data...
3 }
4
5 // This gets called on every request
6 export async function getServerSideProps() {
7   // Fetch data from external API
8   const res = await fetch(`https://.../data`);
9   const data = await res.json();
10
11  // Pass data to the page via props
12  return { props: { data } };
13}
14
15 export default Page;
```

Caching with Server-Side Rendering (SSR)

You can use caching headers (`Cache-Control`) inside `getServerSideProps` to cache dynamic responses. For example, using `stale-while-revalidate` ↗.

```
1 // This value is considered fresh for ten seconds (s-maxage=10).
2 // If a request is repeated within the next 10 seconds, the previously
3 // cached value will still be fresh. If the request is repeated before 59 seconds,
4 // the cached value will be stale but still render (stale-while-revalidate=59).
5 //
6 // In the background, a revalidation request will be made to populate the cache
7 // with a fresh value. If you refresh the page, you will see the new value.
8 export async function getServerSideProps({ req, res }) {
9   res.setHeader(
10     'Cache-Control',
11     'public, s-maxage=10, stale-while-revalidate=59',
12   );
13
14   return {
15     props: {},
16   };
17 }
```

Learn more about [caching](#).

Does `getServerSideProps` render an error page

If an error is thrown inside `getServerSideProps`, it will show the `pages/500.js` file. Check out the documentation for [500 page](#) to learn more on how to create it. During development this file will not be used and the dev overlay will be shown instead.

> Menu

Pages Router > ... > Data Fetching > getStaticPaths

getStaticPaths

If a page has [Dynamic Routes](#) and uses `getStaticProps`, it needs to define a list of paths to be statically generated.

When you export a function called `getStaticPaths` (Static Site Generation) from a page that uses dynamic routes, Next.js will statically pre-render all the paths specified by `getStaticPaths`.

TS pages/repo/[name].tsx

```
1 import type {
2   InferGetStaticPropsType,
3   GetStaticProps,
4   GetStaticPaths,
5 } from 'next';
6
7 type Repo = {
8   name: string;
9   stargazers_count: number;
10};
11
12 export const getStaticPaths: GetStaticPaths = async () => {
13   return {
14     paths: [
15       {
16         params: {
17           name: 'next.js',
18         },
19       }, // See the "paths" section below
20     ],
21     fallback: true, // false or "blocking"
22   };
23 };
24
25 export const getStaticProps: GetStaticProps<{
26   repo: Repo;
27 }> = async () => {
28   const res = await fetch('https://api.github.com/repos/vercel/next.js');
29   const repo = await res.json();
30   return { props: { repo } };
31 }
```

```
32
33 export default function Page({
34   repo,
35 }: InferGetStaticPropsType<typeof getStaticProps>) {
36   return repo.stargazers_count;
37 }
```

The `getStaticPaths` API reference covers all parameters and props that can be used with `getStaticPaths`.

When should I use `getStaticPaths`?

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

- The data comes from a headless CMS
- The data comes from a database
- The data comes from the filesystem
- 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

When does `getStaticPaths` run

`getStaticPaths` will only run during build in production, it will not be called during runtime. You can validate code written inside `getStaticPaths` is removed from the client-side bundle [with this tool ↗](#).

How does `getStaticProps` run with regards to `getStaticPaths`

- `getStaticProps` runs during `next build` for any `paths` returned during build
- `getStaticProps` runs in the background when using `fallback: true`
- `getStaticProps` is called before initial render when using `fallback: blocking`

Where can I use `getStaticPaths`

- `getStaticPaths` **must** be used with `getStaticProps`
- You **cannot** use `getStaticPaths` with `getServerSideProps`
- You can export `getStaticPaths` from a [Dynamic Route](#) that also uses `getStaticProps`
- You **cannot** export `getStaticPaths` from non-page file (e.g. your `components` folder)
- You must export `getStaticPaths` as a standalone function, and not a property of the page component

Runs on every request in development

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

Generating paths on-demand

`getStaticPaths` allows you to control which pages are generated during the build instead of on-demand with `fallback`. Generating more pages during a build will cause slower builds.

You can defer generating all pages on-demand by returning an empty array for `paths`. This can be especially helpful when deploying your Next.js application to multiple environments. For example, you can have faster builds by generating all pages on-demand for previews (but not production builds). This is helpful for sites with hundreds/thousands of static pages.

js pages/posts/[id].js



```
1  export async function getStaticPaths() {  
2      // When this is true (in preview environments) don't  
3      // prerender any static pages  
4      // (faster builds, but slower initial page load)  
5      if (process.env.SKIP_BUILD_STATIC_GENERATION) {  
6          return {  
7              paths: [],  
8              fallback: 'blocking',  
9          };  
10     }  
11  
12     // Call an external API endpoint to get posts  
13     const res = await fetch('https://.../posts');  
14     const posts = await res.json();  
15  
16     // Get the paths we want to prerender based on posts  
17     // In production environments, prerender all pages
```

```
18 // (slower builds, but faster initial page load)
19 const paths = posts.map((post) => ({
20   params: { id: post.id },
21 }));
22
23 // { fallback: false } means other routes should 404
24 return { paths, fallback: false };
25 }
```

> Menu

Pages Router > ... > Data Fetching > getStaticProps

getStaticProps

If you export a function called `getStaticProps` (Static Site Generation) from a page, Next.js will pre-render this page at build time using the props returned by `getStaticProps`.

TS pages/index.tsx

```
1 import type { InferGetStaticPropsType, GetStaticProps } from 'next';
2
3 type Repo = {
4   name: string;
5   stargazers_count: number;
6 };
7
8 export const getStaticProps: GetStaticProps<{
9   repo: Repo;
10 }> = async () => {
11   const res = await fetch('https://api.github.com/repos/vercel/next.js');
12   const repo = await res.json();
13   return { props: { repo } };
14 };
15
16 export default function Page({
17   repo,
18 }: InferGetStaticPropsType<typeof getStaticProps>) {
19   return repo.stargazers_count;
20 }
```

Note that irrespective of rendering type, any `props` will be passed to the page component and can be viewed on the client-side in the initial HTML. This is to allow the page to be [hydrated](#) correctly. Make sure that you don't pass any sensitive information that shouldn't be available on the client in `props`.

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 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
- The data can be publicly cached (not user-specific). This condition can be bypassed in certain specific situations by using a Middleware to rewrite the path.

When does `getStaticProps` run

`getStaticProps` always runs on the server and never on the client. You can validate code written inside `getStaticProps` is removed from the client-side bundle [with this tool ↗](#).

- `getStaticProps` always runs during `next build`
- `getStaticProps` runs in the background when using `fallback: true`
- `getStaticProps` is called before initial render when using `fallback: blocking`
- `getStaticProps` runs in the background when using `revalidate`
- `getStaticProps` runs on-demand in the background when using `revalidate()`

When combined with [Incremental Static Regeneration](#), `getStaticProps` will run in the background while the stale page is being revalidated, and the fresh page served to the browser.

`getStaticProps` does not have access to the incoming request (such as query parameters or HTTP headers) as it generates static HTML. If you need access to the request for your page, consider using [Middleware](#) in addition to `getStaticProps`.

Using `getStaticProps` to fetch data from a CMS

The following example shows how you can fetch a list of blog posts from a CMS.

TS pages/blog.tsx

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

```
3   return (
4     <ul>
5       {posts.map((post) => (
6         <li>{post.title}</li>
7       )))
8     </ul>
9   );
10 }
11
12 // This function gets called at build time on server-side.
13 // It won't be called on client-side, so you can even do
14 // direct database queries.
15 export async function getStaticProps() {
16   // Call an external API endpoint to get posts.
17   // You can use any data fetching library
18   const res = await fetch('https://.../posts');
19   const posts = await res.json();
20
21   // By returning { props: { posts } }, the Blog component
22   // will receive `posts` as a prop at build time
23   return {
24     props: {
25       posts,
26     },
27   };
28 }
```

The `getStaticProps` API reference covers all parameters and props that can be used with `getStaticProps`.

Write server-side code directly

As `getStaticProps` runs only on the server-side, it will never run on the client-side. It won't even be included in the JS bundle for the browser, so you can write direct database queries without them being sent to browsers.

This means that instead of fetching an **API route** from `getStaticProps` (that itself fetches data from an external source), you can write the server-side code directly in `getStaticProps`.

Take the following example. An API route is used to fetch some data from a CMS. That API route is then called directly from `getStaticProps`. This produces an additional call, reducing performance. Instead, the logic for fetching the data from the CMS can be shared by using a `lib/` directory. Then it can be shared with `getStaticProps`.

```
1 // The following function is shared
2 // with getStaticProps and API routes
3 // from a `lib/` directory
4 export async function loadPosts() {
5   // Call an external API endpoint to get posts
6   const res = await fetch('https://.../posts/');
7   const data = await res.json();
8
9   return data;
10}
11
12// pages/blog.js
13import { loadPosts } from '../lib/load-posts';
14
15// This function runs only on the server side
16export async function getStaticProps() {
17  // Instead of fetching your `/api` route you can call the same
18  // function directly in `getStaticProps`
19  const posts = await loadPosts();
20
21  // Props returned will be passed to the page component
22  return { props: { posts } };
23}
```

Alternatively, if you are **not** using API routes to fetch data, then the [fetch\(\)](#) API can be used directly in `getStaticProps` to fetch data.

To verify what Next.js eliminates from the client-side bundle, you can use the [next-code-elimination tool](#).

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](#) or [next/router](#). 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.

When using Incremental Static Generation, `getStaticProps` will be executed in the background to generate the JSON needed for client-side navigation. You may see this in the form of multiple requests being made for the same page, however, this is intended and has no impact on end-user performance.

Where can I use `getStaticProps`

`getStaticProps` can only be exported from a **page**. You **cannot** export it from non-page files, `_app`, `_document`, or `_error`.

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 `getStaticProps` as a standalone function — it will **not** work if you add `getStaticProps` as a property of the page component.

Note: if you have created a [custom app](#), ensure you are passing the `pageProps` to the page component as shown in the linked document, otherwise the props will be empty.

Runs on every request in development

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

Preview Mode

You can temporarily bypass static generation and render the page at **request time** instead of build time using [Preview Mode](#). For example, you might be using a headless CMS and want to preview drafts before they're published.

> Menu

Pages Router > ... > Data Fetching > Incremental Static Regeneration

Incremental Static Regeneration

► Examples

► Version History

Next.js allows you to create or update static pages *after* you've built your site. Incremental Static Regeneration (ISR) enables you to use static-generation on a per-page basis, **without needing to rebuild the entire site**. With ISR, you can retain the benefits of static while scaling to millions of pages.

Note: The `edge runtime` is currently not compatible with ISR, although you can leverage `stale-while-revalidate` by setting the `cache-control` header manually.

To use ISR, add the `revalidate` prop to `getStaticProps`:

```
1 function Blog({ posts }) {
2   return (
3     <ul>
4       {posts.map((post) => (
5         <li key={post.id}>{post.title}</li>
6       )))
7     </ul>
8   );
9 }
10
11 // This function gets called at build time on server-side.
12 // It may be called again, on a serverless function, if
13 // revalidation is enabled and a new request comes in
14 export async function getStaticProps() {
15   const res = await fetch('https://.../posts');
16   const posts = await res.json();
17
18   return {
19     props: {
20       posts,
21     },
22     // Next.js will attempt to re-generate the page:
23     // - When a request comes in
24     // - At most once every 10 seconds
25     revalidate: 10, // In seconds
26   };
27 }
```

```

26   };
27 }
28
29 // This function gets called at build time on server-side.
30 // It may be called again, on a serverless function, if
31 // the path has not been generated.
32 export async function getStaticPaths() {
33   const res = await fetch('https://.../posts');
34   const posts = await res.json();
35
36   // Get the paths we want to pre-render based on posts
37   const paths = posts.map((post) => ({
38     params: { id: post.id },
39   }));
40
41   // We'll pre-render only these paths at build time.
42   // { fallback: 'blocking' } will server-render pages
43   // on-demand if the path doesn't exist.
44   return { paths, fallback: 'blocking' };
45 }
46
47 export default Blog;

```

When a request is made to a page that was pre-rendered at build time, it will initially show the cached page.

- Any requests to the page after the initial request and before 10 seconds are also cached and instantaneous.
- After the 10-second window, the next request will still show the cached (stale) page
- Next.js triggers a regeneration of the page in the background.
- Once the page generates successfully, Next.js will invalidate the cache and show the updated page. If the background regeneration fails, the old page would still be unaltered.

When a request is made to a path that hasn't been generated, Next.js will server-render the page on the first request. Future requests will serve the static file from the cache. ISR on Vercel [persists the cache globally and handles rollbacks ↗](#).

Note: Check if your upstream data provider has caching enabled by default. You might need to disable (e.g. `useCdn: false`), otherwise a revalidation won't be able to pull fresh data to update the ISR cache. Caching can occur at a CDN (for an endpoint being requested) when it returns the `Cache-Control` header.

On-Demand Revalidation

If you set a `revalidate` time of `60`, all visitors will see the same generated version of your site for one minute. The only way to invalidate the cache is from someone visiting that page after the minute has passed.

Starting with `v12.2.0`, Next.js supports On-Demand Incremental Static Regeneration to manually purge the Next.js cache for a specific page. This makes it easier to update your site when:

- Content from your headless CMS is created or updated
- Ecommerce metadata changes (price, description, category, reviews, etc.)

Inside `getStaticProps`, you do not need to specify `revalidate` to use on-demand revalidation. If `revalidate` is omitted, Next.js will use the default value of `false` (no revalidation) and only revalidate the page on-demand when `revalidate()` is called.

Note: [Middleware](#) won't be executed for On-Demand ISR requests. Instead, call `revalidate()` on the exact path that you want revalidated. For example, if you have `pages/blog/[slug].js` and a rewrite from `/post-1 -> /blog/post-1`, you would need to call `res.revalidate('/blog/post-1')`.

Using On-Demand Revalidation

First, create a secret token only known by your Next.js app. This secret will be used to prevent unauthorized access to the revalidation API Route. You can access the route (either manually or with a webhook) with the following URL structure:

>_ Terminal

`https://<your-site.com>/api/revalidate?secret=<token>`

Next, add the secret as an [Environment Variable](#) to your application. Finally, create the revalidation API Route:

js pages/api/revalidate.js

```
1 export default async function handler(req, res) {
2   // Check for secret to confirm this is a valid request
3   if (req.query.secret !== process.env.MY_SECRET_TOKEN) {
4     return res.status(401).json({ message: 'Invalid token' });
5   }
6
7   try {
8     // this should be the actual path not a rewritten path
9     // e.g. for "/blog/[slug]" this should be "/blog/post-1"
10    await res.revalidate('/path-to-revalidate');
```

```
11     return res.json({ revalidated: true });
12 } catch (err) {
13     // If there was an error, Next.js will continue
14     // to show the last successfully generated page
15     return res.status(500).send('Error revalidating');
16 }
17 }
```

View our [demo ↗](#) to see on-demand revalidation in action and provide feedback.

Testing on-Demand ISR during development

When running locally with `next dev`, `getStaticProps` is invoked on every request. To verify your on-demand ISR configuration is correct, you will need to create a [production build](#) and start the [production server](#):

>_ Terminal

```
1 $ next build
2 $ next start
```

Then, you can confirm that static pages have successfully revalidated.

Error handling and revalidation

If there is an error inside `getStaticProps` when handling background regeneration, or you manually throw an error, the last successfully generated page will continue to show. On the next subsequent request, Next.js will retry calling `getStaticProps`.

```
1 export async function getStaticProps() {
2     // If this request throws an uncaught error, Next.js will
3     // not invalidate the currently shown page and
4     // retry getStaticProps on the next request.
5     const res = await fetch('https://.../posts');
6     const posts = await res.json();
7
8     if (!res.ok) {
9         // If there is a server error, you might want to
10        // throw an error instead of returning so that the cache is not updated
11        // until the next successful request.
12        throw new Error(`Failed to fetch posts, received status ${res.status}`);
13    }
14
15    // If the request was successful, return the posts
```

```
16 // and revalidate every 10 seconds.
17 return {
18   props: {
19     posts,
20   },
21   revalidate: 10,
22 };
23 }
```

Self-hosting ISR

Incremental Static Regeneration (ISR) works on [self-hosted Next.js sites](#) out of the box when you use `next start`.

You can use this approach when deploying to container orchestrators such as [Kubernetes ↗](#) or [HashiCorp Nomad ↗](#). By default, generated assets will be stored in-memory on each pod. This means that each pod will have its own copy of the static files. Stale data may be shown until that specific pod is hit by a request.

To ensure consistency across all pods, you can disable in-memory caching. This will inform the Next.js server to only leverage assets generated by ISR in the file system.

You can use a shared network mount in your Kubernetes pods (or similar setup) to reuse the same file-system cache between different containers. By sharing the same mount, the `.next` folder which contains the `next/image` cache will also be shared and re-used.

To disable in-memory caching, set `isrMemoryCacheSize` to `0` in your `next.config.js` file:

JS next.config.js

```
1 module.exports = {
2   experimental: {
3     // Defaults to 50MB
4     isrMemoryCacheSize: 0,
5   },
6 };
```

Note: You might need to consider a race condition between multiple pods trying to update the cache at the same time, depending on how your shared mount is configured.

> Menu

Pages Router > Building Your Application > Deploying

Deploying

Congratulations! You're here because you are ready to deploy your Next.js application. This page will show how to deploy either managed or self-hosted using the [Next.js Build API](#).

Next.js Build API

`next build` generates an optimized version of your application for production. This standard output includes:

- HTML files for pages using `getStaticProps` or [Automatic Static Optimization](#)
- CSS files for global styles or for individually scoped styles
- JavaScript for pre-rendering dynamic content from the Next.js server
- JavaScript for interactivity on the client-side through React

This output is generated inside the `.next` folder:

- `.next/static/chunks/pages` – Each JavaScript file inside this folder relates to the route with the same name. For example, `.next/static/chunks/pages/about.js` would be the JavaScript file loaded when viewing the `/about` route in your application
- `.next/static/media` – Statically imported images from `next/image` are hashed and copied here
- `.next/static/css` – Global CSS files for all pages in your application
- `.next/server/pages` – The HTML and JavaScript entry points prerendered from the server. The `.nft.json` files are created when [Output File Tracing](#) is enabled and contain all the file paths that depend on a given page.
- `.next/server/chunks` – Shared JavaScript chunks used in multiple places throughout your application
- `.next/cache` – Output for the build cache and cached images, responses, and pages from the Next.js server. Using a cache helps decrease build times and improve performance of loading images

All JavaScript code inside `.next` has been **compiled** and browser bundles have been **minified** to help achieve the best performance and support [all modern browsers](#).

Managed Next.js with Vercel

[Vercel](#) ↗ is the fastest way to deploy your Next.js application with zero configuration.

When deploying to Vercel, the platform [automatically detects Next.js](#) ↗, runs `next build`, and optimizes the build output for you, including:

- Persisting cached assets across deployments if unchanged
- [Immutable deployments](#) ↗ with a unique URL for every commit
- [Pages](#) are automatically statically optimized, if possible
- Assets (JavaScript, CSS, images, fonts) are compressed and served from a [Global Edge Network](#) ↗
- [API Routes](#) are automatically optimized as isolated [Serverless Functions](#) ↗ that can scale infinitely
- [Middleware](#) is automatically optimized as [Edge Functions](#) ↗ that have zero cold starts and boot instantly

In addition, Vercel provides features like:

- Automatic performance monitoring with [Next.js Speed Insights](#) ↗
- Automatic HTTPS and SSL certificates
- Automatic CI/CD (through GitHub, GitLab, Bitbucket, etc.)
- Support for [Environment Variables](#) ↗
- Support for [Custom Domains](#) ↗
- Support for [Image Optimization](#) with `next/image`
- Instant global deployments via `git push`

[Deploy a Next.js application to Vercel](#) ↗ for free to try it out.

Self-Hosting

You can self-host Next.js with support for all features using Node.js or Docker. You can also do a Static HTML Export, which [has some limitations](#).

Node.js Server

Next.js can be deployed to any hosting provider that supports Node.js. For example, [AWS EC2 ↗](#) or a [DigitalOcean Droplet ↗](#).

First, ensure your `package.json` has the `"build"` and `"start"` scripts:

```
1  {
2    "scripts": {
3      "dev": "next dev",
4      "build": "next build",
5      "start": "next start"
6    }
7 }
```

Then, run `npm run build` to build your application. Finally, run `npm run start` to start the Node.js server. This server supports all features of Next.js.

If you are using `next/image`, consider adding `sharp` for more performant [Image Optimization](#) in your production environment by running `npm install sharp` in your project directory. On Linux platforms, `sharp` may require [additional configuration ↗](#) to prevent excessive memory usage.

Docker Image

Next.js can be deployed to any hosting provider that supports [Docker ↗](#) containers. You can use this approach when deploying to container orchestrators such as [Kubernetes ↗](#) or [HashiCorp Nomad ↗](#), or when running inside a single node in any cloud provider.

1. [Install Docker ↗](#) on your machine
2. Clone the [with-docker ↗](#) example
3. Build your container: `docker build -t nextjs-docker .`
4. Run your container: `docker run -p 3000:3000 nextjs-docker`

If you need to use different Environment Variables across multiple environments, check out our [with-docker-multi-env ↗](#) example.

Static HTML Export

If you'd like to do a static HTML export of your Next.js app, follow the directions on our [Static HTML Export documentation](#).

Other Services

The following services support Next.js v12+. Below, you'll find examples or guides to deploy Next.js to each service.

Managed Server

- [AWS Copilot ↗](#)
- [Digital Ocean App Platform ↗](#)
- [Google Cloud Run ↗](#)
- [Heroku ↗](#)
- [Railway ↗](#)
- [Render ↗](#)

Note: There are also managed platforms that allow you to use a Dockerfile as shown in the [example above](#).

Static Only

The following services only support deploying Next.js using `output: 'export'`.

- [GitHub Pages ↗](#)

You can also manually deploy the output from `output: 'export'` to any static hosting provider, often through your CI/CD pipeline like GitHub Actions, Jenkins, AWS CodeBuild, Circle CI, Azure Pipelines, and more.

Serverless

- [AWS Amplify ↗](#)
- [Azure Static Web Apps ↗](#)
- [Cloudflare Pages ↗](#)
- [Firebase ↗](#)
- [Netlify ↗](#)
- [Terraform ↗](#)
- [SST ↗](#)

Note: Not all serverless providers implement the [Next.js Build API](#) from `next start`. Please check with the provider to see what features are supported.

Automatic Updates

When you deploy your Next.js application, you want to see the latest version without needing to reload.

Next.js will automatically load the latest version of your application in the background when routing. For client-side navigations, `next/link` will temporarily function as a normal `<a>` tag.

Note: If a new page (with an old version) has already been prefetched by `next/link`, Next.js will use the old version. Navigating to a page that has *not* been prefetched (and is not cached at the CDN level) will load the latest version.

Manual Graceful shutdowns

Sometimes you might want to run some cleanup code on process signals like `SIGTERM` or `SIGINT`.

You can do that by setting the env variable `NEXT_MANUAL_SIG_HANDLE` to `true` and then register a handler for that signal inside your `_document.js` file. Please note that you need to register env variable directly in the system env variable, not in the `.env` file.

package.json

```
1  {
2    "scripts": {
3      "dev": "NEXT_MANUAL_SIG_HANDLE=true next dev",
4      "build": "next build",
5      "start": "NEXT_MANUAL_SIG_HANDLE=true next start"
6    }
7  }
```

pages/_document.js

```
1  if (process.env.NEXT_MANUAL_SIG_HANDLE) {
2    // this should be added in your custom _document
3    process.on('SIGTERM', () => {
4      console.log('Received SIGTERM: ', 'cleaning up');
```

```
5     process.exit(0);
6   });
7
8   process.on('SIGINT', () => {
9     console.log('Received SIGINT: ', 'cleaning up');
10    process.exit(0);
11  });
12 }
```

Going to Production

Before taking your Next.js application to production, here are some recommendations to ensure the best user experience.

Static Exports

Next.js enables starting as a static site or Single-Page Application (SPA), then later optionally upgrading to use features that require a server.

Multi Zones

Learn how to use multi zones to deploy multiple Next.js apps as a single app.

Continuous Integration (CI) Build Caching

Learn how to configure CI to cache Next.js builds

> Menu

Pages Router > ... > Deploying > Continuous Integration (CI) Build Caching

Continuous Integration (CI) Build Caching

To improve build performance, Next.js saves a cache to `.next/cache` that is shared between builds.

To take advantage of this cache in Continuous Integration (CI) environments, your CI workflow will need to be configured to correctly persist the cache between builds.

If your CI is not configured to persist `.next/cache` between builds, you may see a [No Cache Detected](#) error.

Here are some example cache configurations for common CI providers:

Vercel

Next.js caching is automatically configured for you. There's no action required on your part.

CircleCI

Edit your `save_cache` step in `.circleci/config.yml` to include `.next/cache`:

```
1 steps:
2   - save_cache:
3     key: dependency-cache-{{ checksum "yarn.lock" }}
4     paths:
5       - ./node_modules
6       - ./next/cache
```

If you do not have a `save_cache` key, please follow CircleCI's [documentation on setting up build caching ↗](#).

Travis CI

Add or merge the following into your `.travis.yml`:

```
1 cache:  
2   directories:  
3     - $HOME/.cache/yarn  
4     - node_modules  
5     - .next/cache
```

GitLab CI

Add or merge the following into your `.gitlab-ci.yml`:

```
1 cache:  
2   key: ${CI_COMMIT_REF_SLUG}  
3   paths:  
4     - node_modules/  
5     - .next/cache/
```

Netlify CI

Use [Netlify Plugins ↗](#) with [@netlify/plugin-nextjs ↗](#).

AWS CodeBuild

Add (or merge in) the following to your `buildspec.yml`:

```
1 cache:  
2   paths:  
3     - 'node_modules/**/*' # Cache `node_modules` for faster `yarn` or `npm i`  
4     - '.next/cache/**/*' # Cache Next.js for faster application rebuilds
```

GitHub Actions

Using GitHub's [actions/cache](#), add the following step in your workflow file:

```
1 uses: actions/cache@v3
2 with:
3   # See here for caching with `yarn` https://github.com/actions/cache/blob/main/examples#
4   path: |
5     ~/.npm
6     ${{ github.workspace }}/.next/cache
7   # Generate a new cache whenever packages or source files change.
8   key: ${{ runner.os }}-nextjs-${{ hashFiles('**/package-lock.json') }}-${{ hashFiles('**'-
9   # If source files changed but packages didn't, rebuild from a prior cache.
10  restore-keys: |
11    ${{ runner.os }}-nextjs-${{ hashFiles('**/package-lock.json') }}-
```

Bitbucket Pipelines

Add or merge the following into your `bitbucket-pipelines.yml` at the top level (same level as `pipelines`):

```
1 definitions:
2   caches:
3     nextcache: .next/cache
```

Then reference it in the `caches` section of your pipeline's `step`:

```
1 - step:
2   name: your_step_name
3   caches:
4     - node
5     - nextcache
```

Heroku

Using Heroku's [custom cache](#), add a `cacheDirectories` array in your top-level package.json:

```
"cacheDirectories": [ ".next/cache" ]
```

Azure Pipelines

Using Azure Pipelines' [Cache task](#), add the following task to your pipeline yaml file somewhere prior to the task that executes `next build`:

```
1 - task: Cache@2
2   displayName: 'Cache .next/cache'
3   inputs:
4     key: next | $(Agent.OS) | yarn.lock
5     path: '$(System.DefaultWorkingDirectory)/.next/cache'
```

> Menu

Pages Router > ... > Deploying > Multi Zones

Multi Zones

▼ Examples

- [With Zones ↗](#)

A zone is a single deployment of a Next.js app. You can have multiple zones and merge them as a single app.

For example, let's say you have the following apps:

- An app for serving `/blog/**`
- Another app for serving all other pages

With multi zones support, you can merge both these apps into a single one allowing your customers to browse it using a single URL, but you can develop and deploy both apps independently.

How to define a zone

There are no zone related APIs. You only need to do the following:

- Make sure to keep only the pages you need in your app, meaning that an app can't have pages from another app, if app A has `/blog` then app B shouldn't have it too.
- Make sure to configure a [basePath](#) to avoid conflicts with pages and static files.

How to merge zones

You can merge zones using [rewrites](#) in one of the apps or any HTTP proxy.

For [Next.js](#) on [Vercel](#) applications, you can use a [monorepo](#) to deploy both apps with a single `git push`

> Menu

Pages Router > ... > Deploying > Going to Production

Going to Production

Before taking your Next.js application to production, here are some recommendations to ensure the best user experience.

In General

- Use [caching](#) wherever possible.
- Ensure your database and backend are deployed in the same region.
- Aim to ship the least amount of JavaScript possible.
- Defer loading heavy JavaScript bundles until needed.
- Ensure [logging](#) is set up.
- Ensure [error handling](#) is set up.
- Configure the [404](#) (Not Found) and [500](#) (Error) pages.
- Ensure you are [measuring performance](#).
- Run [Lighthouse](#) ↗ to check for performance, best practices, accessibility, and SEO. For best results, use a production build of Next.js and use incognito in your browser so results aren't affected by extensions.
- Review [Supported Browsers and Features](#).
- Improve performance using:
 - [next/image](#) and Automatic Image Optimization
 - [Automatic Font Optimization](#)
 - [Script Optimization](#)
- Improve [loading performance](#)

Caching

▼ Examples

- [ssr-caching ↗](#)

Caching improves response times and reduces the number of requests to external services. Next.js automatically adds caching headers to immutable assets served from `/_next/static` including JavaScript, CSS, static images, and other media.

```
Cache-Control: public, max-age=31536000, immutable
```

`Cache-Control` headers set in `next.config.js` will be overwritten in production to ensure that static assets can be cached effectively. If you need to revalidate the cache of a page that has been [statically generated](#), you can do so by setting `revalidate` in the page's `getStaticProps` function. If you're using `next/image`, you can configure the `minimumCacheTTL` for the default Image Optimization loader.

Note: When running your application locally with `next dev`, your headers are overwritten to prevent caching locally.

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

You can also use caching headers inside `getServerSideProps` and API Routes for dynamic responses. For example, using [stale-while-revalidate ↗](#).

```
1 // This value is considered fresh for ten seconds (s-maxage=10).
2 // If a request is repeated within the next 10 seconds, the previously
3 // cached value will still be fresh. If the request is repeated before 59 seconds,
4 // the cached value will be stale but still render (stale-while-revalidate=59).
5 //
6 // In the background, a revalidation request will be made to populate the cache
7 // with a fresh value. If you refresh the page, you will see the new value.
8 export async function getServerSideProps({ req, res }) {
9   res.setHeader(
10     'Cache-Control',
11     'public, s-maxage=10, stale-while-revalidate=59',
12   );
13
14   return {
15     props: {},
16   };
17 }
```

By default, `Cache-Control` headers will be set differently depending on how your page fetches data.

- If the page uses `getServerSideProps` or `getInitialProps`, it will use the default `Cache-Control` header set by `next start` in order to prevent accidental caching of responses that cannot be cached. If you want a different cache behavior while using `getServerSideProps`, use `res.setHeader('Cache-Control', 'value_you_prefer')` inside of the function as shown above.
- If the page is using `getStaticProps`, it will have a `Cache-Control` header of `s-maxage=REVALIDATE_SECONDS, stale-while-revalidate`, or if `revalidate` is *not* used, `s-maxage=31536000, stale-while-revalidate` to cache for the maximum age possible.

Note: Your deployment provider must support caching for dynamic responses. If you are self-hosting, you will need to add this logic yourself using a key/value store like Redis. If you are using Vercel, [Edge Caching works without configuration ↗](#).

Reducing JavaScript Size

▼ Examples

- [with-dynamic-import ↗](#)

To reduce the amount of JavaScript sent to the browser, you can use the following tools to understand what is included inside each JavaScript bundle:

- [Import Cost ↗](#) – Display the size of the imported package inside VSCode.
- [Package Phobia ↗](#) – Find the cost of adding a new dev dependency to your project.
- [Bundle Phobia ↗](#) - Analyze how much a dependency can increase bundle sizes.
- [Webpack Bundle Analyzer ↗](#) – Visualize the size of webpack output files with an interactive, zoomable treemap.
- [bundlejs ↗](#) - An online tool to quickly bundle & minify your projects, while viewing the compressed gzip/brotli bundle size, all running locally on your browser.

Each file inside your `pages/` directory will automatically be code split into its own JavaScript bundle during `next build`. You can also use [Dynamic Imports](#) to lazy-load components and libraries. For example, you might want to defer loading your modal code until a user clicks the open button.

Logging

▼ Examples

- [Pino and Logflare Example ↗](#)

Since Next.js runs on both the client and server, there are multiple forms of logging supported:

- `console.log` in the browser
- `stdout` on the server

If you want a structured logging package, we recommend [Pino ↗](#). If you're using Vercel, there are [pre-built logging integrations ↗](#) compatible with Next.js.

Error Handling

▼ Examples

- [with-sentry ↗](#)

When an unhandled exception occurs, you can control the experience for your users with the [500 page](#). We recommend customizing this to your brand instead of the default Next.js theme.

You can also log and track exceptions with a tool like Sentry. [This example ↗](#) shows how to catch & report errors on both the client and server-side, using the Sentry SDK for Next.js. There's also a [Sentry integration for Vercel ↗](#).

Loading Performance

To improve loading performance, you first need to determine what to measure and how to measure it. [Core Web Vitals ↗](#) is a good industry standard that is measured using your own web browser. If you are not familiar with the metrics of Core Web Vitals, review this [blog post ↗](#) and determine which specific metric/s will be your drivers for loading performance. Ideally, you would want to measure the loading performance in the following environments:

- In the lab, using your own computer or a simulator.

- In the field, using real-world data from actual visitors.
- Local, using a test that runs on your device.
- Remote, using a test that runs in the cloud.

Once you are able to measure the loading performance, use the following strategies to improve it iteratively so that you apply one strategy, measure the new performance and continue tweaking until you do not see much improvement. Then, you can move on to the next strategy.

- Use caching regions that are close to the regions where your database or API is deployed.
- As described in the [caching](#) section, use a `stale-while-revalidate` value that will not overload your backend.
- Use [Incremental Static Regeneration](#) to reduce the number of requests to your backend.
- Remove unused JavaScript. Review this [blog post ↗](#) to understand what Core Web Vitals metrics bundle size affects and what strategies you can use to reduce it, such as:
 - Setting up your Code Editor to view import costs and sizes
 - Finding alternative smaller packages
 - Dynamically loading components and dependencies

> Menu

Pages Router > ... > Deploying > Static Exports

Static Exports

Next.js enables starting as a static site or Single-Page Application (SPA), then later optionally upgrading to use features that require a server.

When running `next build`, Next.js generates an HTML file per route. By breaking a strict SPA into individual HTML files, Next.js can avoid loading unnecessary JavaScript code on the client-side, reducing the bundle size and enabling faster page loads.

Since Next.js supports this static export, it can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets.

Note: We recommend using the App Router for enhanced static export support.

Configuration

To enable a static export, change the output mode inside `next.config.js`:

JS next.config.js

```
1  /**
2   * @type {import('next').NextConfig}
3   */
4  const nextConfig = {
5    output: 'export',
6    // Optional: Add a trailing slash to all paths `/about` -> `/about/`
7    // trailingSlash: true,
8    // Optional: Change the output directory `out` -> `dist`
9    // distDir: 'dist',
10 };
11
12 module.exports = nextConfig;
```

After running `next build`, Next.js will produce an `out` folder which contains the HTML/CSS/JS assets for your application.

You can utilize `getStaticProps` and `getStaticPaths` to generate an HTML file for each page in your `pages` directory (or more for [dynamic routes](#)).

Supported Features

The majority of core Next.js features needed to build a static site are supported, including:

- [Dynamic Routes when using `getStaticPaths`](#)
- Prefetching with `next/link`
- Preloading JavaScript
- [Dynamic Imports](#)
- Any styling options (e.g. CSS Modules, styled-jsx)
- [Client-side data fetching](#)
- `getStaticProps`
- `getStaticPaths`

Image Optimization

[Image Optimization](#) through `next/image` can be used with a static export by defining a custom image loader in `next.config.js`. For example, you can optimize images with a service like Cloudinary:

`js` `next.config.js`

```
1  /** @type {import('next').NextConfig} */
2  const nextConfig = {
3    output: 'export',
4    images: {
5      loader: 'custom',
6      loaderFile: './app/image.ts',
7    },
8  };
9
10 module.exports = nextConfig;
```

This custom loader will define how to fetch images from a remote source. For example, the following loader will construct the URL for Cloudinary:

```
1 export default function cloudinaryLoader({
2   src,
3   width,
4   quality,
5 }: {
6   src: string;
7   width: number;
8   quality?: number;
9 }) {
10   const params = ['f_auto', 'c_limit', `w_${width}`, `q_${quality || 'auto'}`];
11   return `https://res.cloudinary.com/demo/image/upload/${params.join(
12     ',',
13   )}${src}`;
14 }
```

You can then use `next/image` in your application, defining relative paths to the image in Cloudinary:

```
1 import Image from 'next/image';
2
3 export default function Page() {
4   return <Image alt="turtles" src="/turtles.jpg" width={300} height={300} />;
5 }
```

Unsupported Features

Features that require a Node.js server, or dynamic logic that cannot be computed during the build process, are not supported:

- [Internationalized Routing](#)
- [API Routes](#)
- [Rewrites](#)
- [Redirects](#)
- [Headers](#)
- [Middleware](#)
- [Incremental Static Regeneration](#)
- [getStaticPaths with fallback: true](#)

- `getStaticPaths` with `fallback: 'blocking'`
 - `getServerSideProps`
 - [Image Optimization](#) (default loader)
-

Deploying

With a static export, Next.js can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets.

When running `next build`, Next.js generates the static export into the `out` folder. Using `next export` is no longer needed. For example, let's say you have the following routes:

- `/`
- `/blog/[id]`

After running `next build`, Next.js will generate the following files:

- `/out/index.html`
- `/out/404.html`
- `/out/blog/post-1.html`
- `/out/blog/post-2.html`

If you are using a static host like Nginx, you can configure rewrites from incoming requests to the correct files:

```
nginx.conf
```

```
1 server {
2   listen 80;
3   server_name acme.com;
4
5   root /var/www;
6
7   location / {
8     try_files /out/index.html =404;
9   }
10
11  location /blog/ {
12    rewrite ^/blog/(.*)$ /out/blog/$1.html break;
13  }
14}
```

```
15     error_page 404 /out/404.html;
16     location = /404.html {
17         internal;
18     }
19 }
```

Version History

Version Changes

v13.4.0 App Router (Stable) adds enhanced static export support, including using React Server Components and Route Handlers

v13.3.0 `next export` is deprecated and replaced with `"output": "export"`

> Menu

Pages Router > Building Your Application > Optimizing

Optimizations

Next.js comes with a variety of built-in optimizations designed to improve your application's speed and [Core Web Vitals ↗](#). This guide will cover the optimizations you can leverage to enhance your user experience.

Built-in Components

Built-in components abstract away the complexity of implementing common UI optimizations. These components are:

- **Images:** Built on the native `` element. The Image Component optimizes images for performance by lazy loading and automatically resizing images based on device size.
- **Link:** Built on the native `<a>` tags. The Link Component prefetches pages in the background, for faster and smoother page transitions.
- **Scripts:** Built on the native `<script>` tags. The Script Component gives you control over loading and execution of third-party scripts.

Metadata

Metadata helps search engines understand your content better (which can result in better SEO), and allows you to customize how your content is presented on social media, helping you create a more engaging and consistent user experience across various platforms.

The Head Component in Next.js allows you to modify the `<head>` of a page. Learn more in the [Head Component](#) documentation.

Static Assets

Next.js `/public` folder can be used to serve static assets like images, fonts, and other files. Files inside `/public` can also be cached by CDN providers so that they are delivered efficiently.

Analytics and Monitoring

For large applications, Next.js integrates with popular analytics and monitoring tools to help you understand how your application is performing. Learn more in the [Analytics](#), [OpenTelemetry](#), and [Instrumentation](#) guides.

Images

Optimize your images with the built-in `'next/image'` component.

Fonts

Optimize your application's web fonts with the built-in `'next/font'` loaders.

Scripts

Optimize 3rd party scripts with the built-in `Script` component.

Static Assets

Next.js allows you to serve static files, like images, in the `public` directory. You can learn how it works [here](#).

Lazy Loading

Lazy load imported libraries and React Components to improve your application's loading performance.

Analytics

Measure and track page performance using Next.js Speed Insights

OpenTelemetry

Learn how to instrument your Next.js app with OpenTelemetry.

Instrumentation

Learn how to use instrumentation to run code at server startup in your Next.js app

Testing

Learn how to set up Next.js with three commonly used testing tools — Cypress, Playwright, Jest, and React Testing Library.

> Menu

Pages Router > ... > Optimizing > Analytics

Analytics

Next.js Speed Insights [↗](#) allows you to analyze and measure the performance of pages using different metrics.

You can start collecting your Real Experience Score [↗](#) with zero-configuration on Vercel deployments [↗](#).

The rest of this documentation describes the built-in relayer Next.js Speed Insights uses.

Build Your Own

First, you will need to create a [custom App](#) component and define a `reportWebVitals` function:

js pages/_app.js

```
1 export function reportWebVitals(metric) {
2   console.log(metric);
3 }
4
5 function MyApp({ Component, pageProps }) {
6   return <Component {...pageProps} />;
7 }
8
9 export default MyApp;
```



This function is fired when the final values for any of the metrics have finished calculating on the page. You can use to log any of the results to the console or send to a particular endpoint.

The `metric` object returned to the function consists of a number of properties:

- `id`: Unique identifier for the metric in the context of the current page load
- `name`: Metric name

- `startTime` : First recorded timestamp of the performance entry in [milliseconds ↗](#) (if applicable)
- `value` : Value, or duration in [milliseconds ↗](#), of the performance entry
- `label` : Type of metric (`web-vital` or `custom`)

There are two types of metrics that are tracked:

- Web Vitals
 - Custom metrics
-

Web Vitals

[Web Vitals ↗](#) are a set of useful metrics that aim to capture the user experience of a web page. The following web vitals are all included:

- [Time to First Byte ↗ \(TTFB\)](#)
- [First Contentful Paint ↗ \(FCP\)](#)
- [Largest Contentful Paint ↗ \(LCP\)](#)
- [First Input Delay ↗ \(FID\)](#)
- [Cumulative Layout Shift ↗ \(CLS\)](#)
- [Interaction to Next Paint ↗ \(INP\) \(experimental\)](#)

You can handle all the results of these metrics using the `web-vital` label:

```
1 export function reportWebVitals(metric) {  
2   if (metric.label === 'web-vital') {  
3     console.log(metric); // The metric object ({ id, name, startTime, value, label }) is  
4   }  
5 }
```

There's also the option of handling each of the metrics separately:

```
1 export function reportWebVitals(metric) {  
2   switch (metric.name) {  
3     case 'FCP':  
4       // handle FCP results  
5       break;  
6     case 'LCP':  
7       // handle LCP results  
8       break;
```

```
9  case 'CLS':
10     // handle CLS results
11     break;
12 case 'FID':
13     // handle FID results
14     break;
15 case 'TTFB':
16     // handle TTFB results
17     break;
18 case 'INP':
19     // handle INP results (note: INP is still an experimental metric)
20     break;
21 default:
22     break;
23 }
24 }
```

A third-party library, [web-vitals](#), is used to measure these metrics. Browser compatibility depends on the particular metric, so refer to the [Browser Support](#) section to find out which browsers are supported.

Custom metrics

In addition to the core metrics listed above, there are some additional custom metrics that measure the time it takes for the page to hydrate and render:

- `Next.js-hydration`: Length of time it takes for the page to start and finish hydrating (in ms)
- `Next.js-route-change-to-render`: Length of time it takes for a page to start rendering after a route change (in ms)
- `Next.js-render`: Length of time it takes for a page to finish render after a route change (in ms)

You can handle all the results of these metrics using the `custom` label:

```
1 export function reportWebVitals(metric) {
2   if (metric.label === 'custom') {
3     console.log(metric); // The metric object ({ id, name, startTime, value, label }) is
4   }
5 }
```

There's also the option of handling each of the metrics separately:

```
1 export function reportWebVitals(metric) {
2   switch (metric.name) {
3     case 'Next.js-hydration':
```

```
4     // handle hydration results
5     break;
6     case 'Next.js-route-change-to-render':
7         // handle route-change to render results
8         break;
9     case 'Next.js-render':
10        // handle render results
11        break;
12    default:
13        break;
14    }
15 }
```

These metrics work in all browsers that support the [User Timing API](#).

Sending results to external systems

With the relay function, you can send results to any endpoint to measure and track real user performance on your site. For example:

```
1 export function reportWebVitals(metric) {
2   const body = JSON.stringify(metric);
3   const url = 'https://example.com/analytics';
4
5   // Use `navigator.sendBeacon()` if available, falling back to `fetch()``.
6   if (navigator.sendBeacon) {
7     navigator.sendBeacon(url, body);
8   } else {
9     fetch(url, { body, method: 'POST', keepalive: true });
10  }
11 }
```

Note: If you use [Google Analytics](#), using the `id` value can allow you to construct metric distributions manually (to calculate percentiles, etc.)

```
1 export function reportWebVitals({ id, name, label, value }) {
2   // Use `window.gtag` if you initialized Google Analytics as this example:
3   // https://github.com/vercel/next.js/blob/canary/examples/with-google-analytics/page
4   window.gtag('event', name, {
5     event_category:
6       label === 'web-vital' ? 'Web Vitals' : 'Next.js custom metric',
7     value: Math.round(name === 'CLS' ? value * 1000 : value), // values must be integers
8     event_label: id, // id unique to current page load
9     non_interaction: true, // avoids affecting bounce rate.
10   });
11 }
```

Read more about [sending results to Google Analytics](#).

TypeScript

If you are using TypeScript, you can use the built-in type `NextWebVitalsMetric`:

pages/_app.tsx



```
1 import type { AppProps, NextWebVitalsMetric } from 'next/app';
2
3 function MyApp({ Component, pageProps }: AppProps) {
4   return <Component {...pageProps} />;
5 }
6
7 export function reportWebVitals(metric: NextWebVitalsMetric) {
8   console.log(metric);
9 }
10
11 export default MyApp;
```

> Menu

Pages Router > ... > Optimizing > Fonts

Font Optimization

`next/font` will automatically optimize your fonts (including custom fonts) and remove external network requests for improved privacy and performance.

Watch: Learn more about how to use `next/font` → [YouTube \(6 minutes\)](#) ↗.

`next/font` includes **built-in automatic self-hosting** for *any* font file. This means you can optimally load web fonts with zero layout shift, thanks to the underlying CSS `size-adjust` property used.

This new font system also allows you to conveniently use all Google Fonts with performance and privacy in mind. CSS and font files are downloaded at build time and self-hosted with the rest of your static assets. **No requests are sent to Google by the browser.**

Google Fonts

Automatically self-host any Google Font. Fonts are included in the deployment and served from the same domain as your deployment. **No requests are sent to Google by the browser.**

Get started by importing the font you would like to use from `next/font/google` as a function. We recommend using [variable fonts](#) ↗ for the best performance and flexibility.

To use the font in all your pages, add it to `_app.js` file under `/pages` as shown below:

js pages/_app.js



```
1 import { Inter } from 'next/font/google';
2
3 // If loading a variable font, you don't need to specify the font weight
4 const inter = Inter({ subsets: ['latin'] });
5
6 export default function MyApp({ Component, pageProps }) {
```

```
7   return (
8     <main className={inter.className}>
9       <Component {...pageProps} />
10    </main>
11  );
12 }
```

If you can't use a variable font, you will **need to specify a weight**:

js pages/_app.js

```
1 import { Roboto } from 'next/font/google';
2
3 const roboto = Roboto({
4   weight: '400',
5   subsets: ['latin'],
6 });
7
8 export default function MyApp({ Component, pageProps }) {
9   return (
10     <main className={roboto.className}>
11       <Component {...pageProps} />
12     </main>
13   );
14 }
```

You can specify multiple weights and/or styles by using an array:

js app/layout.js

```
1 const roboto = Roboto({
2   weight: ['400', '700'],
3   style: ['normal', 'italic'],
4   subsets: ['latin'],
5   display: 'swap',
6 });
```

Good to know: Use an underscore (_) for font names with multiple words. E.g. `Roboto Mono` should be imported as `Roboto_Mono`.

Apply the font in `<head>`

You can also use the font without a wrapper and `className` by injecting it inside the `<head>` as follows:

js pages/_app.js

```
1 import { Inter } from 'next/font/google';
2
3 const inter = Inter({ subsets: ['latin'] });
4
5 export default function MyApp({ Component, pageProps }) {
6   return (
7     <>
8       <style jsx global>{
9         html {
10           font-family: ${inter.style.fontFamily};
11         }
12       `}</style>
13       <Component {...pageProps} />
14     </>
15   );
16 }
```

Single page usage

To use the font on a single page, add it to the specific page as shown below:

js pages/index.js

```
1 import { Inter } from 'next/font/google';
2
3 const inter = Inter({ subsets: ['latin'] });
4
5 export default function Home() {
6   return (
7     <div className={inter.className}>
8       <p>Hello World</p>
9     </div>
10  );
11 }
```

Specifying a subset

Google Fonts are automatically [subset ↗](#). This reduces the size of the font file and improves performance. You'll need to define which of these subsets you want to preload. Failing to specify any subsets while `preload` is `true` will result in a warning.

This can be done by adding it to the function call:

js pages/_app.js

```
const inter = Inter({ subsets: ['latin'] });
```

Using Multiple Fonts

You can import and use multiple fonts in your application. There are two approaches you can take.

The first approach is to create a utility function that exports a font, imports it, and applies its `className` where needed. This ensures the font is preloaded only when it's rendered:

app/fonts.ts

```
1 import { Inter, Roboto_Mono } from 'next/font/google';
2
3 export const inter = Inter({
4   subsets: ['latin'],
5   display: 'swap',
6 });
7
8 export const roboto_mono = Roboto_Mono({
9   subsets: ['latin'],
10  display: 'swap',
11});
```

In the example above, `Inter` will be applied globally, and `Roboto Mono` can be imported and applied as needed.

Alternatively, you can create a [CSS variable](#) and use it with your preferred CSS solution:

app/global.css

```
1 html {
2   font-family: var(--font-inter);
3 }
4
5 h1 {
6   font-family: var(--font-roboto-mono);
7 }
```

In the example above, `Inter` will be applied globally, and any `<h1>` tags will be styled with `Roboto Mono`.

Recommendation: Use multiple fonts conservatively since each new font is an additional resource the client has to download.

Local Fonts

Import `next/font/local` and specify the `src` of your local font file. We recommend using [variable fonts](#) for the best performance and flexibility.

JS pages/_app.js

```
1 import localFont from 'next/font/local';
2
3 // Font files can be colocated inside of `pages`
4 const myFont = localFont({ src: './my-font.woff2' });
5
6 export default function MyApp({ Component, pageProps }) {
7   return (
8     <main className={myFont.className}>
9       <Component {...pageProps} />
10      </main>
11    );
12 }
```

If you want to use multiple files for a single font family, `src` can be an array:

```
1 const roboto = localFont({
2   src: [
3     {
4       path: './Roboto-Regular.woff2',
5       weight: '400',
6       style: 'normal',
7     },
8     {
9       path: './Roboto-Italic.woff2',
10      weight: '400',
11      style: 'italic',
12    },
13    {
14      path: './Roboto-Bold.woff2',
15      weight: '700',
16      style: 'normal',
17    },
18    {
19      path: './Roboto-BoldItalic.woff2',
20      weight: '700',
21      style: 'italic',
22    },
23  ],
24});
```

With Tailwind CSS

`next/font` can be used with [Tailwind CSS](#) through a [CSS variable](#).

In the example below, we use the font `Inter` from `next/font/google` (you can use any font from Google or Local Fonts). Load your font with the `variable` option to define your CSS variable name and assign it to `inter`. Then, use `inter.variable` to add the CSS variable to your HTML document.

js pages/_app.js

```
1 import { Inter } from 'next/font/google';
2
3 const inter = Inter({
4   subsets: ['latin'],
5   variable: '--font-inter',
6 });
7
8 export default function MyApp({ Component, pageProps }) {
9   return (
10     <main className={`${inter.variable} font-sans`}>
11       <Component {...pageProps} />
12     </main>
13   );
14 }
```

Finally, add the CSS variable to your [Tailwind CSS config](#):

tailwind.config.js

```
1 /** @type {import('tailwindcss').Config} */
2 module.exports = {
3   content: [
4     './pages/**/*.{js,ts,jsx,tsx}',
5     './components/**/*.{js,ts,jsx,tsx}',
6   ],
7   theme: {
8     extend: {
9       fontFamily: {
10         sans: ['var(--font-inter)'],
11         mono: ['var(--font-roboto-mono)'],
12       },
13     },
14   },
15   plugins: [],
}
```

You can now use the `font-sans` and `font-mono` utility classes to apply the font to your elements.

Preloading

When a font function is called on a page of your site, it is not globally available and preloaded on all routes. Rather, the font is only preloaded on the related route/s based on the type of file where it is used:

- if it's a [unique page](#), it is preloaded on the unique route for that page
- if it's in the [custom App](#), it is preloaded on all the routes of the site under `/pages`

Reusing fonts

Every time you call the `localFont` or Google font function, that font is hosted as one instance in your application. Therefore, if you load the same font function in multiple files, multiple instances of the same font are hosted. In this situation, it is recommended to do the following:

- Call the font loader function in one shared file
- Export it as a constant
- Import the constant in each file where you would like to use this font

> Menu

Pages Router > ... > Optimizing > Images

Image Optimization

► Examples

The Next.js Image component extends the HTML `` element with:

- **Size Optimization:** Automatically serve correctly sized images for each device, using modern image formats like WebP and AVIF.
- **Visual Stability:** Prevent [layout shift](#) automatically when images are loading.
- **Faster Page Loads:** Images are only loaded when they enter the viewport using native browser lazy loading, with optional blur-up placeholders.
- **Asset Flexibility:** On-demand image resizing, even for images stored on remote servers

Usage

```
import Image from 'next/image';
```

You can then define the `src` for your image (either local or remote).

Local Images

To use a local image, `import` your `.jpg`, `.png`, or `.webp` image files.

Next.js will [automatically determine](#) the `width` and `height` of your image based on the imported file. These values are used to prevent [Cumulative Layout Shift](#) while your image is loading.

JS pages/index.js



```
1 import Image from 'next/image';
2 import profilePic from '../public/me.png';
```

```
3
4 export default function Page() {
5   return (
6     <Image
7       src={profilePic}
8       alt="Picture of the author"
9       // width={500} automatically provided
10      // height={500} automatically provided
11      // blurDataURL="data:..." automatically provided
12      // placeholder="blur" // Optional blur-up while loading
13    />
14  );
15}
```

Warning: Dynamic `await import()` or `require()` are *not* supported. The `import` must be static so it can be analyzed at build time.

Remote Images

To use a remote image, the `src` property should be a URL string.

Since Next.js does not have access to remote files during the build process, you'll need to provide the `width`, `height` and optional `blurDataURL` props manually.

The `width` and `height` attributes are used to infer the correct aspect ratio of image and avoid layout shift from the image loading in. The `width` and `height` do *not* determine the rendered size of the image file. Learn more about [Image Sizing](#).

js app/page.js

```
1 import Image from 'next/image';
2
3 export default function Page() {
4   return (
5     <Image
6       src="https://s3.amazonaws.com/my-bucket/profile.png"
7       alt="Picture of the author"
8       width={500}
9       height={500}
10    />
11  );
12}
```

To safely allow optimizing images, define a list of supported URL patterns in `next.config.js`. Be as specific as possible to prevent malicious usage. For example, the following configuration will only allow images from a specific AWS S3 bucket:



```
1 module.exports = {
2   images: {
3     remotePatterns: [
4       {
5         protocol: 'https',
6         hostname: 's3.amazonaws.com',
7         port: '',
8         pathname: '/my-bucket/**',
9       },
10    ],
11  },
12};
```

Learn more about [remotePatterns](#) configuration. If you want to use relative URLs for the image `src`, use a [loader](#).

Domains

Sometimes you may want to optimize a remote image, but still use the built-in Next.js Image Optimization API. To do this, leave the [loader](#) at its default setting and enter an absolute URL for the Image `src` prop.

To protect your application from malicious users, you must define a list of remote hostnames you intend to use with the [next/image](#) component.

Learn more about [remotePatterns](#) configuration.

Loaders

Note that in the [example earlier](#), a partial URL (`"/me.png"`) is provided for a remote image. This is possible because of the loader architecture.

A loader is a function that generates the URLs for your image. It modifies the provided `src`, and generates multiple URLs to request the image at different sizes. These multiple URLs are used in the automatic [srcset](#) generation, so that visitors to your site will be served an image that is the right size for their viewport.

The default loader for Next.js applications uses the built-in Image Optimization API, which optimizes images from anywhere on the web, and then serves them directly from the Next.js web server. If you would like to serve your images directly from a CDN or image server, you can write your own loader function with a few lines of JavaScript.

You can define a loader per-image with the `loader` prop, or at the application level with the `loaderFile` configuration.

Priority

You should add the `priority` property to the image that will be the Largest Contentful Paint (LCP) element ↗ for each page. Doing so allows Next.js to specially prioritize the image for loading (e.g. through preload tags or priority hints), leading to a meaningful boost in LCP.

The LCP element is typically the largest image or text block visible within the viewport of the page. When you run `next dev`, you'll see a console warning if the LCP element is an `<Image>` without the `priority` property.

Once you've identified the LCP image, you can add the property like this:

JS app/page.js

```
1 import Image from 'next/image';
2
3 export default function Home() {
4   return (
5     <>
6       <h1>My Homepage</h1>
7       <Image
8         src="/me.png"
9         alt="Picture of the author"
10        width={500}
11        height={500}
12        priority
13      />
14      <p>Welcome to my homepage!</p>
15    </>
16  );
17}
```

See more about priority in the [next/image component documentation](#).

Image Sizing

One of the ways that images most commonly hurt performance is through *layout shift*, where the image pushes other elements around on the page as it loads in. This performance problem is so annoying to users that it has its own Core Web Vital, called [Cumulative Layout Shift ↗](#). The way to avoid image-based layout shifts is to [always size your images ↗](#). This allows the browser to reserve precisely enough space for the image before it loads.

Because `next/image` is designed to guarantee good performance results, it cannot be used in a way that will contribute to layout shift, and **must** be sized in one of three ways:

1. Automatically, using a [static import](#)
2. Explicitly, by including a `width` and `height` property
3. Implicitly, by using `fill` which causes the image to expand to fill its parent element.

What if I don't know the size of my images?

If you are accessing images from a source without knowledge of the images' sizes, there are several things you can do:

Use `fill`

The `fill` prop allows your image to be sized by its parent element. Consider using CSS to give the image's parent element space on the page along `sizes` prop to match any media query break points. You can also use `object-fit` ↗ with `fill`, `contain`, or `cover`, and `object-position` ↗ to define how the image should occupy that space.

Normalize your images

If you're serving images from a source that you control, consider modifying your image pipeline to normalize the images to a specific size.

Modify your API calls

If your application is retrieving image URLs using an API call (such as to a CMS), you may be able to modify the API call to return the image dimensions along with the URL.

If none of the suggested methods works for sizing your images, the `next/image` component is designed to work well on a page alongside standard `` elements.

Styling

Styling the Image component is similar to styling a normal `` element, but there are a few guidelines to keep in mind:

- Use `className` or `style`, not `styled-jsx`.
 - In most cases, we recommend using the `className` prop. This can be an imported [CSS Module](#), a [global stylesheet](#), etc.

- You can also use the `style` prop to assign inline styles.
- You cannot use `styled-jsx` because it's scoped to the current component (unless you mark the style as `global`).
- When using `fill`, the parent element must have `position: relative`
 - This is necessary for the proper rendering of the image element in that layout mode.
- When using `fill`, the parent element must have `display: block`
 - This is the default for `<div>` elements but should be specified otherwise.

For examples, see the [Image Component Demo ↗](#).

Examples

For examples of the Image component used with the various styles, see the [Image Component Demo ↗](#).

Other Properties

[View all properties available to the `next/image` component.](#)

Configuration

The `next/image` component and Next.js Image Optimization API can be configured in the `next.config.js` file. These configurations allow you to [enable remote images](#), [define custom image breakpoints](#), [change caching behavior](#) and more.

[Read the full image configuration documentation for more information.](#)

> Menu

Pages Router > ... > Optimizing > Instrumentation

Instrumentation

Note: This feature is experimental. To use it, you must explicitly opt in by defining `experimental.instrumentationHook = true;` in your `next.config.js`.

If you export a function named `register` from this file, we will call that function whenever a new Next.js server instance is bootstrapped. When your `register` function is deployed, it will be called on each cold boot (but exactly once in each environment).

Sometimes, it may be useful to import a file in your code because of the side effects it will cause. For example, you might import a file that defines a set of global variables, but never explicitly use the imported file in your code. You would still have access to the global variables the package has declared.

You can import files with side effects in `instrumentation.ts`, which you might want to use in your `register` function as demonstrated in the following example:

ts /instrumentation.ts

```
1 import { init } from 'package-init';
2
3 export function register() {
4   init();
5 }
```

However, we recommend importing files with side effects using `import` from within your `register` function instead. The following example demonstrates a basic usage of `import` in a `register` function:

ts /instrumentation.ts

```
1 export async function register() {
2   await import('package-with-side-effect');
3 }
```

By doing this, you can colocate all of your side effects in one place in your code, and avoid any unintended consequences from importing files.

We call `register` in all environments, so it's necessary to conditionally import any code that doesn't support both `edge` and `nodejs`. You can use the environment variable `NEXT_RUNTIME` to get the current environment. Importing an environment-specific code would look like this:

TS /instrumentation.ts

```
1 export async function register() {
2   if (process.env.NEXT_RUNTIME === 'nodejs') {
3     await import('./instrumentation-node');
4   }
5
6   if (process.env.NEXT_RUNTIME === 'edge') {
7     await import('./instrumentation-edge');
8   }
9 }
```

> Menu

Pages Router > ... > Optimizing > Lazy Loading

Lazy Loading

[Lazy loading ↗](#) in Next.js helps improve the initial loading performance of an application by decreasing the amount of JavaScript needed to render a route.

It allows you to defer loading of **Client Components** and imported libraries, and only include them in the client bundle when they're needed. For example, you might want to defer loading a modal until a user clicks to open it.

There are two ways you can implement lazy loading in Next.js:

1. Using [Dynamic Imports](#) with `next/dynamic`
2. Using `React.lazy()` ↗ with [Suspense ↗](#)

By default, Server Components are automatically [code split ↗](#), and you can use [streaming](#) to progressively send pieces of UI from the server to the client. Lazy loading applies to Client Components.

next/dynamic

`next/dynamic` is a composite of [React.lazy\(\)](#) ↗ and [Suspense ↗](#). It behaves the same way in the `app` and `pages` directories to allow for incremental migration.

Examples

By using `next/dynamic`, the header component will not be included in the page's initial JavaScript bundle. The page will render the Suspense `fallback` first, followed by the `Header` component when the `Suspense` boundary is resolved.

```
1 import dynamic from 'next/dynamic';
2
3 const DynamicHeader = dynamic(() => import('../components/header'), {
4   loading: () => <p>Loading...</p>,
5 });
6
7 export default function Home() {
8   return <DynamicHeader />;
9 }
```

Note: In `import('path/to/component')`, the path must be explicitly written. It can't be a template string nor a variable. Furthermore the `import()` has to be inside the `dynamic()` call for Next.js to be able to match webpack bundles / module ids to the specific `dynamic()` call and preload them before rendering. `dynamic()` can't be used inside of React rendering as it needs to be marked in the top level of the module for preloading to work, similar to `React.lazy`.

With named exports

To dynamically import a named export, you can return it from the `Promise` returned by `import()`:

JS components/hello.js

```
1 export function Hello() {
2   return <p>Hello!</p>;
3 }
4
5 // pages/index.js
6 import dynamic from 'next/dynamic';
7
8 const DynamicComponent = dynamic(() =>
9   import('../components/hello').then((mod) => mod.Hello),
10 );
```

With no SSR

To dynamically load a component on the client side, you can use the `ssr` option to disable server-rendering. This is useful if an external dependency or component relies on browser APIs like `window`.

```
1 import dynamic from 'next/dynamic';
```

```
2
3 const DynamicHeader = dynamic(() => import('../components/header'), {
4   ssr: false,
5 });

```

With external libraries

This example uses the external library `fuse.js` for fuzzy search. The module is only loaded in the browser after the user types in the search input.

```
1 import { useState } from 'react';
2
3 const names = ['Tim', 'Joe', 'Bel', 'Lee'];
4
5 export default function Page() {
6   const [results, setResults] = useState();
7
8   return (
9     <div>
10       <input
11         type="text"
12         placeholder="Search"
13         onChange={async (e) => {
14           const { value } = e.currentTarget;
15           // Dynamically load fuse.js
16           const Fuse = (await import('fuse.js')).default;
17           const fuse = new Fuse(names);
18
19           setResults(fuse.search(value));
20         }}
21       />
22       <pre>Results: {JSON.stringify(results, null, 2)}</pre>
23     </div>
24   );
25 }
```

> Menu

Pages Router > ... > Optimizing > OpenTelemetry

OpenTelemetry

Note: This feature is experimental, you need to explicitly opt-in by providing `experimental.instrumentationHook = true;` in your `next.config.js`.

Observability is crucial for understanding and optimizing the behavior and performance of your Next.js app.

As applications become more complex, it becomes increasingly difficult to identify and diagnose issues that may arise. By leveraging observability tools, such as logging and metrics, developers can gain insights into their application's behavior and identify areas for optimization. With observability, developers can proactively address issues before they become major problems and provide a better user experience. Therefore, it is highly recommended to use observability in your Next.js applications to improve performance, optimize resources, and enhance user experience.

We recommend using OpenTelemetry for instrumenting your apps. It's a platform-agnostic way to instrument apps that allows you to change your observability provider without changing your code. Read [Official OpenTelemetry docs ↗](#) for more information about OpenTelemetry and how it works.

This documentation uses terms like *Span*, *Trace* or *Exporter* throughout this doc, all of which can be found in [the OpenTelemetry Observability Primer ↗](#).

Next.js supports OpenTelemetry instrumentation out of the box, which means that we already instrumented Next.js itself. When you enable OpenTelemetry we will automatically wrap all your code like `getStaticProps` in *spans* with helpful attributes.

Note: We currently support OpenTelemetry bindings only in serverless functions. We don't provide any for `edge` or client side code.

Getting Started

OpenTelemetry is extensible but setting it up properly can be quite verbose. That's why we prepared a package `@vercel/otel` that helps you get started quickly. It's not extensible and you should configure OpenTelemetry manually you need to customize your setup.

Using `@vercel/otel`

To get started, you must install `@vercel/otel`:

```
>_ Terminal
```

```
npm install @vercel/otel
```

Next, create a custom `instrumentation.ts` file in the root of the project:

```
TS instrumentation.ts
```

```
1 import { registerOTel } from '@vercel/otel';
2
3 export function register() {
4   registerOTel('next-app');
5 }
```

Note: We have created a basic [with-opentelemetry](#) example that you can use.

Manual OpenTelemetry configuration

If our wrapper `@vercel/otel` doesn't suit your needs, you can configure OpenTelemetry manually.

Firstly you need to install OpenTelemetry packages:

```
>_ Terminal
```

```
npm install @opentelemetry/sdk-node @opentelemetry/resources @opentelemetry/semantic-conventi
```

Now you can initialize `NodeSDK` in your `instrumentation.ts`. OpenTelemetry APIs are not compatible with edge runtime, so you need to make sure that you are importing them only when `process.env.NEXT_RUNTIME === 'nodejs'`. We recommend creating a new file `instrumentation.node.ts` which you conditionally import only when using node:

```
TS instrumentation.ts
```

```
1 export async function register() {
2   if (process.env.NEXT_RUNTIME === 'nodejs') {
3     await import('./instrumentation.node.ts');
4   }
5 }
```

ts instrumentation.node.ts

```
1 import { trace, context } from '@opentelemetry/api';
2 import { NodeSDK } from '@opentelemetry/sdk-node';
3 import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http';
4 import { Resource } from '@opentelemetry/resources';
5 import { SemanticResourceAttributes } from '@opentelemetry/semantic-conventions';
6 import { SimpleSpanProcessor } from '@opentelemetry/sdk-trace-node';
7
8 const sdk = new NodeSDK({
9   resource: new Resource({
10     [SemanticResourceAttributes.SERVICE_NAME]: 'next-app',
11   }),
12   spanProcessor: new SimpleSpanProcessor(new OTLPTraceExporter()),
13 });
14 sdk.start();
```

Doing this is equivalent to using `@vercel/otel`, but it's possible to modify and extend. For example, you could use `@opentelemetry/exporter-trace-otlp-grpc` instead of `@opentelemetry/exporter-trace-otlp-http` or you can specify more resource attributes.

Testing your instrumentation

You need an OpenTelemetry collector with a compatible backend to test OpenTelemetry traces locally. We recommend using our [OpenTelemetry dev environment ↗](#).

If everything works well you should be able to see the root server span labeled as `GET /requested pathname`. All other spans from that particular trace will be nested under it.

Next.js traces more spans than are emitted by default. To see more spans, you must set `NEXT_OTEL_VERBOSE=1`.

Deployment

Using OpenTelemetry Collector

When you are deploying with OpenTelemetry Collector, you can use `@vercel/otel`. It will work both on Vercel and when self-hosted.

Deploying on Vercel

We made sure that OpenTelemetry works out of the box on Vercel.

Follow [Vercel documentation](#) ↗ to connect your project to an observability provider.

Self-hosting

Deploying to other platforms is also straightforward. You will need to spin up your own OpenTelemetry Collector to receive and process the telemetry data from your Next.js app.

To do this, follow the [OpenTelemetry Collector Getting Started guide](#) ↗, which will walk you through setting up the collector and configuring it to receive data from your Next.js app.

Once you have your collector up and running, you can deploy your Next.js app to your chosen platform following their respective deployment guides.

Custom Exporters

We recommend using OpenTelemetry Collector. If that is not possible on your platform, you can use a custom OpenTelemetry exporter with [manual OpenTelemetry configuration](#)

Custom Spans

You can add a custom span with [OpenTelemetry APIs](#) ↗.

```
>_ Terminal
```

```
npm install @opentelemetry/api
```

The following example demonstrates a function that fetches GitHub stars and adds a custom `fetchGithubStars` span to track the fetch request's result:

```
1 import { trace } from '@opentelemetry/api';
2
3 export async function fetchGithubStars() {
4     return await trace
```

```
5     .getTracer('nextjs-example')
6     .startActiveSpan('fetchGithubStars', async (span) => {
7       try {
8         return await getValue();
9       } finally {
10        span.end();
11      }
12    });
13 }
```

The `register` function will execute before your code runs in a new environment. You can start creating new spans, and they should be correctly added to the exported trace.

Default Spans in Next.js

Next.js automatically instruments several spans for you to provide useful insights into your application's performance.

Attributes on spans follow [OpenTelemetry semantic conventions](#). We also add some custom attributes under the `next` namespace:

- `next.span_name` - duplicates span name
- `next.span_type` - each span type has a unique identifier
- `next.route` - The route pattern of the request (e.g., `/[param]/user`).
- `next.page`
 - This is an internal value used by an app router.
 - You can think about it as a route to a special file (like `page.ts`, `layout.ts`, `loading.ts` and others)
 - It can be used as a unique identifier only when paired with `next.route` because `/layout` can be used to identify both `/(groupA)/layout.ts` and `/(groupB)/layout.ts`

[`http.method`] [`next.route`]

- `next.span_type`: `BaseServer.handleRequest`

This span represents the root span for each incoming request to your Next.js application. It tracks the HTTP method, route, target, and status code of the request.

Attributes:

- [Common HTTP attributes ↗](#)

- `http.method`
- `http.status_code`

- [Server HTTP attributes ↗](#)

- `http.route`
 - `http.target`
- `next.span_name`
 - `next.span_type`
 - `next.route`

`render route (app) [next.route]`

- `next.span_type`: `AppRender.getBodyResult`.

This span represents the process of rendering a route in the app router.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`fetch [http.method] [http.url]`

- `next.span_type`: `AppRender.fetch`

This span represents the fetch request executed in your code.

Attributes:

- [Common HTTP attributes ↗](#)
 - `http.method`
 - [Client HTTP attributes ↗](#)
 - `http.url`
 - `net.peer.name`
 - `net.peer.port` (only if specified)
- `next.span_name`

- `next.span_type`

executing api route (app) [next.route]

- `next.span_type`: `AppRouteRouteHandlers.runHandler`.

This span represents the execution of an API route handler in the app router.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

getServerSideProps [next.route]

- `next.span_type`: `Render.getServerSideProps`.

This span represents the execution of `getServerSideProps` for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

getStaticProps [next.route]

- `next.span_type`: `Render.getStaticProps`.

This span represents the execution of `getStaticProps` for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

render route (pages) [next.route]

- `next.span_type`: `Render.renderDocument`.

This span represents the process of rendering the document for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

generateMetadata [next.page]

- `next.span_type`: `ResolveMetadata.generateMetadata`.

This span represents the process of generating metadata for a specific page (a single route can have multiple of these spans).

Attributes:

- `next.span_name`
- `next.span_type`
- `next.page`

> Menu

Pages Router > ... > Optimizing > Scripts

Script Optimization

Application Scripts

To load a third-party script for all routes, import `next/script` and include the script directly in your custom `_app`:

```
1 import Script from 'next/script';
2
3 export default function MyApp({ Component, pageProps }) {
4   return (
5     <>
6       <Component {...pageProps} />
7       <Script src="https://example.com/script.js" />
8     </>
9   );
10 }
```

This script will load and execute when *any* route in your application is accessed. Next.js will ensure the script will **only load once**, even if a user navigates between multiple pages.

Recommendation: We recommend only including third-party scripts in specific pages or layouts in order to minimize any unnecessary impact to performance.

Strategy

Although the default behavior of `next/script` allows you load third-party scripts in any page or layout, you can fine-tune its loading behavior by using the `strategy` property:

- `beforeInteractive` : Load the script before any Next.js code and before any page hydration occurs.
- `afterInteractive` : (**default**) Load the script early but after some hydration on the page occurs.
- `lazyOnload` : Load the script later during browser idle time.
- `worker` : (experimental) Load the script in a web worker.

Refer to the [next/script](#) API reference documentation to learn more about each strategy and their use cases.

Offloading Scripts To A Web Worker (Experimental)

Warning: The `worker` strategy is not yet stable and does not yet work with the `app` directory. Use with caution.

Scripts that use the `worker` strategy are offloaded and executed in a web worker with [Partytown](#). This can improve the performance of your site by dedicating the main thread to the rest of your application code.

This strategy is still experimental and can only be used if the `nextScriptWorkers` flag is enabled in `next.config.js`:

JS next.config.js

```
1 module.exports = {
2   experimental: {
3     nextScriptWorkers: true,
4   },
5 }
```

Then, run `next` (normally `npm run dev` or `yarn dev`) and Next.js will guide you through the installation of the required packages to finish the setup:

> Terminal

```
npm run dev
```

You'll see instructions like these: Please install Partytown by running

```
npm install @builder.io/partytown
```

Once setup is complete, defining `strategy="worker"` will automatically instantiate Partytown in your application and offload the script to a web worker.

TS pages/home.tsx

```
1 import Script from 'next/script';
2
3 export default function Home() {
4   return (
5     <>
6       <Script src="https://example.com/script.js" strategy="worker" />
7     </>
8   )
9 }
```

```
8    );
9 }
```

There are a number of trade-offs that need to be considered when loading a third-party script in a web worker. Please see Partytown's [tradeoffs](#) documentation for more information.

Inline Scripts

Inline scripts, or scripts not loaded from an external file, are also supported by the Script component. They can be written by placing the JavaScript within curly braces:

```
1 <Script id="show-banner">
2   {`document.getElementById('banner').classList.remove('hidden')`}
3 </Script>
```

Or by using the `dangerouslySetInnerHTML` property:

```
1 <Script
2   id="show-banner"
3   dangerouslySetInnerHTML={{
4     __html: `document.getElementById('banner').classList.remove('hidden')`,
5   }}
6 />
```

Warning: An `id` property must be assigned for inline scripts in order for Next.js to track and optimize the script.

Executing Additional Code

Event handlers can be used with the Script component to execute additional code after a certain event occurs:

- `onLoad` : Execute code after the script has finished loading.
- `onReady` : Execute code after the script has finished loading and every time the component is mounted.
- `onError` : Execute code if the script fails to load.

These handlers will only work when `next/script` is imported and used inside of a [Client Component](#) where `"use client"` is defined as the first line of code:

pages/index.tsx

```
1 import Script from 'next/script';
```

```
2
3 export default function Page() {
4   return (
5     <>
6       <Script
7         src="https://example.com/script.js"
8         onLoad={() => {
9           console.log('Script has loaded');
10          }}
11        />
12      </>
13    );
14 }
```

Refer to the [next/script](#) API reference to learn more about each event handler and view examples.

Additional Attributes

There are many DOM attributes that can be assigned to a `<script>` element that are not used by the Script component, like [nonce](#) ↗ or [custom data attributes](#) ↗. Including any additional attributes will automatically forward it to the final, optimized `<script>` element that is included in the HTML.

pages/index.tsx

```
1 import Script from 'next/script';
2
3 export default function Page() {
4   return (
5     <>
6       <Script
7         src="https://example.com/script.js"
8         id="example-script"
9         nonce="XUENAJFW"
10        data-test="script"
11      />
12      </>
13    );
14 }
```

> Menu

Pages Router > ... > Optimizing > Static Assets

Static Assets

Next.js can serve static files, like images, under a folder called `public` in the root directory. Files inside `public` can then be referenced by your code starting from the base URL (`/`).

For example, if you add `me.png` inside `public`, the following code will access the image:

```
1 import Image from 'next/image';
2
3 function Avatar() {
4   return <Image src="/me.png" alt="me" width="64" height="64" />;
5 }
6
7 export default Avatar;
```

This folder is also useful for `robots.txt`, `favicon.ico`, Google Site Verification, and any other static files (including `.html`)!

- Be sure the directory is named `public`. The name cannot be changed and is the only directory used to serve static assets.
- Be sure to not have a static file with the same name as a file in the `pages/` directory, as this will result in an error. [Read more](#)
- Only assets that are in the `public` directory at `build time` will be served by Next.js. Files added at runtime won't be available. We recommend using a third party service like [AWS S3](#) for persistent file storage.

> Menu

Pages Router > ... > Optimizing > Testing

Testing

▼ Examples

- [Next.js with Cypress ↗](#)
- [Next.js with Playwright ↗](#)
- [Next.js with Jest and React Testing Library ↗](#)
- [Next.js with Vitest ↗](#)

Learn how to set up Next.js with commonly used testing tools: [Cypress](#), [Playwright](#), and [Jest with React Testing Library](#).

Cypress

Cypress is a test runner used for **End-to-End (E2E)** and **Component Testing**.

Quickstart

You can use `create-next-app` with the [with-cypress example ↗](#) to quickly get started.

> Terminal



```
npx create-next-app@latest --example with-cypress with-cypress-app
```

Manual setup

To get started with Cypress, install the `cypress` package:

> Terminal



```
npm install --save-dev cypress
```

Add Cypress to the `package.json` scripts field:

```
1 "scripts": {  
2   "dev": "next dev",  
3   "build": "next build",  
4   "start": "next start",  
5   "cypress": "cypress open",  
6 }
```

Run Cypress for the first time to generate examples that use their recommended folder structure:

>_ Terminal

```
npm run cypress
```

You can look through the generated examples and the [Writing Your First Test ↗](#) section of the Cypress Documentation to help you get familiar with Cypress.

Should I use E2E or Component Tests?

The [Cypress docs contain a guide ↗](#) on the difference between these two types of tests and when it is appropriate to use each.

Creating your first Cypress E2E test

Assuming the following two Next.js pages:

js pages/index.js

```
1 import Link from 'next/link';  
2  
3 export default function Home() {  
4   return (  
5     <nav>  
6       <h1>Homepage</h1>  
7       <Link href="/about">About</Link>  
8     </nav>  
9   );  
10 }
```



```

1 export default function About() {
2   return (
3     <div>
4       <h1>About Page</h1>
5       <Link href="/">Homepage</Link>
6     </div>
7   );
8 }

```

Add a test to check your navigation is working correctly:



```

1 describe('Navigation', () => {
2   it('should navigate to the about page', () => {
3     // Start from the index page
4     cy.visit('http://localhost:3000/');
5
6     // Find a link with an href attribute containing "about" and click it
7     cy.get('a[href*="about"]').click();
8
9     // The new url should include "/about"
10    cy.url().should('include', '/about');
11
12    // The new page should contain an h1 with "About page"
13    cy.get('h1').contains('About Page');
14  });
15 });

```

You can use `cy.visit("/")` instead of `cy.visit("http://localhost:3000/")` if you add `baseUrl: 'http://localhost:3000'` to the `cypress.config.js` configuration file.

Creating your first Cypress component test

Component tests build and mount a specific component without having to bundle your whole application or launch a server. This allows for more performant tests that still provide visual feedback and the same API used for Cypress E2E tests.

Note: Since component tests do not launch a Next.js server, capabilities like `<Image />` and `getServerSideProps` which rely on a server being available will not function out-of-the-box. See the [Cypress Next.js docs](#) for examples of getting these features working within component tests.

Assuming the same components from the previous section, add a test to validate a component is rendering the expected output:



```

1 import AboutPage from './about.js';
2
3 describe('<AboutPage />', () => {
4   it('should render and display expected content', () => {
5     // Mount the React component for the About page
6     cy.mount(<AboutPage />);
7
8     // The new page should contain an h1 with "About page"
9     cy.get('h1').contains('About Page');
10
11    // Validate that a link with the expected URL is present
12    // *Following* the link is better suited to an E2E test
13    cy.get('a[href="/"]').should('be.visible');
14  });
15});
```

Running your Cypress tests

E2E Tests

Since Cypress E2E tests are testing a real Next.js application they require the Next.js server to be running prior to starting Cypress. We recommend running your tests against your production code to more closely resemble how your application will behave.

Run `npm run build` and `npm run start`, then run `npm run cypress -- --e2e` in another terminal window to start Cypress and run your E2E testing suite.

Note: Alternatively, you can install the `start-server-and-test` package and add it to the `package.json` scripts field: `"test": "start-server-and-test start http://localhost:3000 cypress"` to start the Next.js production server in conjunction with Cypress. Remember to rebuild your application after new changes.

Component Tests

Run `npm run cypress -- --component` to start Cypress and execute your component testing suite.

Getting ready for Continuous Integration (CI)

You will have noticed that running Cypress so far has opened an interactive browser which is not ideal for CI environments. You can also run Cypress headlessly using the `cypress run` command:



```
3 //...
4 "e2e": "start-server-and-test dev http://localhost:3000 \"cypress open --e2e\"",
5 "e2e:headless": "start-server-and-test dev http://localhost:3000 \"cypress run --e2e\""
6 "component": "cypress open --component",
7 "component:headless": "cypress run --component"
8 }
```

You can learn more about Cypress and Continuous Integration from these resources:

- [Cypress Continuous Integration Docs ↗](#)
- [Cypress GitHub Actions Guide ↗](#)
- [Official Cypress GitHub Action ↗](#)

Playwright

Playwright is a testing framework that lets you automate Chromium, Firefox, and WebKit with a single API. You can use it to write **End-to-End (E2E)** and **Integration** tests across all platforms.

Quickstart

The fastest way to get started is to use `create-next-app` with the [with-playwright example ↗](#). This will create a Next.js project complete with Playwright all set up.

>_ Terminal



```
npx create-next-app@latest --example with-playwright with-playwright-app
```

Manual setup

You can also use `npm init playwright` to add Playwright to an existing [NPM](#) project.

To manually get started with Playwright, install the `@playwright/test` package:

>_ Terminal



```
npm install --save-dev @playwright/test
```

Add Playwright to the `package.json` scripts field:

```
1 "scripts": {
2   "dev": "next dev",
3   "build": "next build",
4   "start": "next start",
5   "test:e2e": "playwright test",
6 }
```

Creating your first Playwright end-to-end test

Assuming the following two Next.js pages:

JS pages/index.js

```
1 import Link from 'next/link';
2
3 export default function Home() {
4   return (
5     <nav>
6       <Link href="/about">About</Link>
7     </nav>
8   );
9 }
```

JS pages/about.js

```
1 export default function About() {
2   return (
3     <div>
4       <h1>About Page</h1>
5     </div>
6   );
7 }
```

Add a test to verify that your navigation is working correctly:

TS e2e/example.spec.ts

```
1 import { test, expect } from '@playwright/test';
2
3 test('should navigate to the about page', async ({ page }) => {
4   // Start from the index page (the baseURL is set via the webServer in the playwright.co
5   await page.goto('http://localhost:3000/');
6   // Find an element with the text 'About Page' and click on it
7   await page.click('text=About');
8   // The new URL should be "/about" (baseURL is used there)
9   await expect(page).toHaveURL('http://localhost:3000/about');
10  // The new page should contain an h1 with "About Page"
```

```
11     await expect(page.locator('h1')).toContainText('About Page');
12 });


```

You can use `page.goto("/")` instead of `page.goto("http://localhost:3000/")`, if you add `"baseURL": "http://localhost:3000"` [↗](#) to the `playwright.config.ts` configuration file.

Running your Playwright tests

Since Playwright is testing a real Next.js application, it requires the Next.js server to be running prior to starting Playwright. It is recommended to run your tests against your production code to more closely resemble how your application will behave.

Run `npm run build` and `npm run start`, then run `npm run test:e2e` in another terminal window to run the Playwright tests.

Note: Alternatively, you can use the `webServer` [↗](#) feature to let Playwright start the development server and wait until it's fully available.

Running Playwright on Continuous Integration (CI)

Playwright will by default run your tests in the [headless mode](#) [↗](#). To install all the Playwright dependencies, run `npx playwright install-deps`.

You can learn more about Playwright and Continuous Integration from these resources:

- [Getting started with Playwright](#) [↗](#)
- [Use a development server](#) [↗](#)
- [Playwright on your CI provider](#) [↗](#)

Jest and React Testing Library

Jest and React Testing Library are frequently used together for **Unit Testing**. There are three ways you can start using Jest within your Next.js application:

1. Using one of our [quickstart examples](#)
2. With the [Next.js Rust Compiler](#)
3. With [Babel](#)

The following sections will go through how you can set up Jest with each of these options:

Quickstart

You can use `create-next-app` with the [with-jest](#) example to quickly get started with Jest and React Testing Library:

>_ Terminal

```
npx create-next-app@latest --example with-jest with-jest-app
```

Setting up Jest (with the Rust Compiler)

Since the release of [Next.js 12](#), Next.js now has built-in configuration for Jest.

To set up Jest, install `jest`, `jest-environment-jsdom`, `@testing-library/react`, `@testing-library/jest-dom`:

>_ Terminal

```
npm install --save-dev jest jest-environment-jsdom @testing-library/react @testing-library/jest-dom
```

Create a `jest.config.mjs` file in your project's root directory and add the following:

js jest.config.mjs

```
1 import nextJest from 'next/jest';
2
3 const createJestConfig = nextJest({
4   // Provide the path to your Next.js app to load next.config.js and .env files in your t
5   dir: './',
6 });
7
8 // Add any custom config to be passed to Jest
9 /** @type {import('jest').Config} */
10 const config = {
11   // Add more setup options before each test is run
12   // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
13
14   testEnvironment: 'jest-environment-jsdom',
15 };
16
17 // createJestConfig is exported this way to ensure that next/jest can load the Next.js co
18 export default createJestConfig(config);
```

Under the hood, `next/jest` is automatically configuring Jest for you, including:

- Setting up `transform` using [SWC](#)
- Auto mocking stylesheets (`.css`, `.module.css`, and their scss variants), image imports and [next/font](#)
- Loading `.env` (and all variants) into `process.env`
- Ignoring `node_modules` from test resolving and transforms
- Ignoring `.next` from test resolving
- Loading `next.config.js` for flags that enable SWC transforms

Note: To test environment variables directly, load them manually in a separate setup script or in your `jest.config.js` file. For more information, please see [Test Environment Variables](#).

Setting up Jest (with Babel)

If you opt out of the [Rust Compiler](#), you will need to manually configure Jest and install `babel-jest` and `identity-obj-proxy` in addition to the packages above.

Here are the recommended options to configure Jest for Next.js:

`js` `jest.config.js`



```
1 module.exports = {
2   collectCoverage: true,
3   // on node 14.x coverage provider v8 offers good speed and more or less good report
4   coverageProvider: 'v8',
5   collectCoverageFrom: [
6     '**/*.{js,jsx,ts,tsx}',
7     '!**/*.d.ts',
8     '!**/node_modules/**',
9     '!<rootDir>/out/**',
10    '!<rootDir>/.next/**',
11    '!<rootDir>/*.config.js',
12    '!<rootDir>/coverage/**',
13  ],
14  moduleNameMapper: {
15    // Handle CSS imports (with CSS modules)
16    // https://jestjs.io/docs/webpack#mocking-css-modules
17    '^.+\\\\.module\\\\.(css|sass|scss)$': 'identity-obj-proxy',
18
19    // Handle CSS imports (without CSS modules)
20    '^.+\\\\.\\.(css|sass|scss)$': '<rootDir>/_mocks_/styleMock.js',
21
22    // Handle image imports
23    // https://jestjs.io/docs/webpack#handling-static-assets
```

```
24  '^.+\\.(png|jpg|jpeg|gif|webp|avif|ico|bmp|svg)$': `<rootDir>/__mocks__/fileMock.js`  
25  
26  // Handle module aliases  
27  '^@/components/(.*)$': '<rootDir>/components/$1',  
28 },  
29 // Add more setup options before each test is run  
30 // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],  
31 testPathIgnorePatterns: ['<rootDir>/node_modules/', '<rootDir>/.next/'],  
32 testEnvironment: 'jsdom',  
33 transform: {  
    // Use babel-jest to transpile tests with the next/babel preset  
    // https://jestjs.io/docs/configuration#transform-objectstring-pathtotransformer--pat  
36  '^.+\\.(js|jsx|ts|tsx)$': ['babel-jest', { presets: ['next/babel'] }],  
37 },  
38 transformIgnorePatterns: [  
    '/node_modules/',  
    '^.+\\.(module\\.(css|sass|scss))$',  
41 ],  
42 };
```

You can learn more about each configuration option in the [Jest docs](#) ↗.

Handling stylesheets and image imports

Stylesheets and images aren't used in the tests but importing them may cause errors, so they will need to be mocked. Create the mock files referenced in the configuration above - `fileMock.js` and `styleMock.js` - inside a `__mocks__` directory:

`JS` `__mocks__/fileMock.js` ✖

```
1 module.exports = {  
2   src: '/img.jpg',  
3   height: 24,  
4   width: 24,  
5   blurDataURL: 'data:image/png;base64,imagedata',  
6 };
```

`JS` `__mocks__/styleMock.js` ✖

```
module.exports = {};
```

For more information on handling static assets, please refer to the [Jest Docs](#) ↗.

Optional: Extend Jest with custom matchers

`@testing-library/jest-dom` includes a set of convenient [custom matchers](#) ↗ such as `.toBeInTheDocument()` making it easier to write tests. You can import the custom matchers for every test

by adding the following option to the Jest configuration file:

```
js jest.config.js
```

```
setupFilesAfterEnv: ['<rootDir>/jest.setup.js'];
```

Then, inside `jest.setup.js`, add the following import:

```
js jest.setup.js
```

```
import '@testing-library/jest-dom/extend-expect';
```

If you need to add more setup options before each test, it's common to add them to the `jest.setup.js` file above.

Optional: Absolute Imports and Module Path Aliases

If your project is using [Module Path Aliases](#), you will need to configure Jest to resolve the imports by matching the paths option in the `jsconfig.json` file with the `moduleNameMapper` option in the `jest.config.js` file. For example:

```
json tsconfig.json
```

```
1  {
2    "compilerOptions": {
3      "baseUrl": ".",
4      "paths": {
5        "@/components/*": ["components/*"]
6      }
7    }
8 }
```

```
js jest.config.js
```

```
1 moduleNameMapper: {
2   '^@/components/(.*)$': '<rootDir>/components/$1',
3 }
```

Creating your tests:

Add a test script to `package.json`

Add the Jest executable in watch mode to the `package.json` scripts:

```
1 "scripts": {  
2   "dev": "next dev",  
3   "build": "next build",  
4   "start": "next start",  
5   "test": "jest --watch"  
6 }
```

`jest --watch` will re-run tests when a file is changed. For more Jest CLI options, please refer to the [Jest Docs](#).

Create your first tests

Your project is now ready to run tests. Follow Jest's convention by adding tests to the `__tests__` folder in your project's root directory.

For example, we can add a test to check if the `<Home />` component successfully renders a heading:

`__tests__/index.test.jsx`

```
1 import { render, screen } from '@testing-library/react';  
2 import Home from '../pages/index';  
3 import '@testing-library/jest-dom';  
4  
5 describe('Home', () => {  
6   it('renders a heading', () => {  
7     render(<Home />);  
8  
9     const heading = screen.getByRole('heading', {  
10       name: /welcome to next\.js!/i,  
11     });  
12  
13     expect(heading).toBeInTheDocument();  
14   });  
15 });
```

Optionally, add a [snapshot test](#) to keep track of any unexpected changes to your `<Home />` component:

`__tests__/snapshot.js`

```
1 import { render } from '@testing-library/react';  
2 import Home from '../pages/index';  
3  
4 it('renders homepage unchanged', () => {  
5   const { container } = render(<Home />);  
6   expect(container).toMatchSnapshot();
```

```
7 });
```

Note: Test files should not be included inside the Pages Router because any files inside the Pages Router are considered routes.

Running your test suite

Run `npm run test` to run your test suite. After your tests pass or fail, you will notice a list of interactive Jest commands that will be helpful as you add more tests.

For further reading, you may find these resources helpful:

- [Jest Docs ↗](#)
- [React Testing Library Docs ↗](#)
- [Testing Playground ↗](#) - use good testing practices to match elements.

Community Packages and Examples

The Next.js community has created packages and articles you may find helpful:

- [next-router-mock ↗](#) for Storybook.
- [Test Preview Vercel Deploys with Cypress ↗](#) by Gleb Bahmutov.

For more information on what to read next, we recommend:

- [pages/basic-features/environment-variables#test-environment-variables](#)

Rendering

By default, Next.js **pre-renders** every page. This means that Next.js generates HTML for each page in advance, instead of having it all done by client-side JavaScript. Pre-rendering can result in better performance and SEO.

Each generated HTML is associated with minimal JavaScript code necessary for that page. When a page is loaded by the browser, its JavaScript code runs and makes the page fully interactive (this process is called [hydration ↗](#) in React).

Pre-rendering

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: The HTML is generated at **build time** and will be reused on each request.
- Server-side Rendering: The HTML is generated on **each request**.

Importantly, Next.js lets you choose which pre-rendering form you'd like 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.

We recommend using Static Generation over Server-side Rendering for performance reasons. Statically generated pages can be cached by CDN with no extra configuration to boost performance. However, in some cases, Server-side Rendering might be the only option.

You can also use client-side data fetching along with Static Generation or Server-side Rendering. That means some parts of a page can be rendered entirely by clientside JavaScript. To learn more, take a look at the [Data Fetching](#) documentation.

Use Server-side Rendering to render pages on each request.

Static Site Generation (SSG)

Use Static Site Generation (SSG) to pre-render pages at build time.

Incremental Static Regeneration (ISR)

Learn how to create or update static pages at runtime with Incremental Static Regeneration.

Automatic Static Optimization

Next.js automatically optimizes your app to be static HTML whenever possible. Learn how it works here.

Client-side Rendering (CSR)

Learn how to implement client-side rendering with Next.js.

Edge and Node.js Runtimes

Learn more about the switchable runtimes (Edge and Node.js) in Next.js.

> Menu

Pages Router > ... > Rendering > Automatic Static Optimization

Automatic Static Optimization

Next.js automatically determines that a page is static (can be prerendered) if it has no blocking data requirements. This determination is made by the absence of `getServerSideProps` and `getInitialProps` in the page.

This feature allows Next.js to emit hybrid applications that contain **both server-rendered and statically generated pages**.

Statically generated pages are still reactive: Next.js will hydrate your application client-side to give it full interactivity.

One of the main benefits of this feature is that optimized pages require no server-side computation, and can be instantly streamed to the end-user from multiple CDN locations. The result is an *ultra fast* loading experience for your users.

How it works

If `getServerSideProps` or `getInitialProps` is present in a page, Next.js will switch to render the page on-demand, per-request (meaning [Server-Side Rendering](#)).

If the above is not the case, Next.js will **statically optimize** your page automatically by prerendering the page to static HTML.

During prerendering, the router's `query` object will be empty since we do not have `query` information to provide during this phase. After hydration, Next.js will trigger an update to your application to provide the route parameters in the `query` object.

The cases where the query will be updated after hydration triggering another render are:

- The page is a [dynamic-route](#).

- The page has query values in the URL.
- [Rewrites](#) are configured in your `next.config.js` since these can have parameters that may need to be parsed and provided in the `query`.

To be able to distinguish if the query is fully updated and ready for use, you can leverage the `isReady` field on `next/router`.

Note: Parameters added with [dynamic routes](#) to a page that's using `getStaticProps` will always be available inside the `query` object.

`next build` will emit `.html` files for statically optimized pages. For example, the result for the page `pages/about.js` would be:

> Terminal

`.next/server/pages/about.html`

And if you add `getServerSideProps` to the page, it will then be JavaScript, like so:

> Terminal

`.next/server/pages/about.js`

Caveats

- If you have a [custom App](#) with `getInitialProps` then this optimization will be turned off in pages without [Static Generation](#).
- If you have a [custom Document](#) with `getInitialProps` be sure you check if `ctx.req` is defined before assuming the page is server-side rendered. `ctx.req` will be `undefined` for pages that are prerendered.
- Avoid using the `asPath` value on `next/router` in the rendering tree until the router's `isReady` field is `true`. Statically optimized pages only know `asPath` on the client and not the server, so using it as a prop may lead to mismatch errors. The [active-class-name example ↗](#) demonstrates one way to use `asPath` as a prop.

> Menu

Pages Router > ... > Rendering > Client-side Rendering (CSR)

Client-side Rendering (CSR)

This page is a work in progress.

> Menu

Pages Router > ... > Rendering > Edge and Node.js Runtimes

Edge and Node.js Runtimes

In the context of Next.js, "runtime" refers to the set of libraries, APIs, and general functionality available to your code during execution.

Next.js has two server runtimes where you can render parts of your application code:

- [Node.js Runtime](#)
- [Edge Runtime](#)

Each runtime has its own set of APIs. Please refer to the [Node.js Docs ↗](#) and [Edge Docs](#) for the full list of available APIs. Both runtimes can also support [streaming](#) depending on your deployment infrastructure.

By default, the `app` directory uses the Node.js runtime. However, you can opt into different runtimes (e.g. Edge) on a per-route basis.

Runtime Differences

There are many considerations to make when choosing a runtime. This table shows the major differences at a glance. If you want a more in-depth analysis of the differences, check out the sections below.

	Node	Serverless	Edge
Cold Boot ↗	/	~250ms	Instant
HTTP Streaming	Yes	Yes	Yes
IO	All	All	<code>fetch</code>
Scalability	/	High	Highest
Security	Normal	High	High
Latency	Normal	Low	Lowest

Node	Serverless	Edge
npm Packages	All	A smaller subset

Edge Runtime

In Next.js, the lightweight Edge Runtime is a subset of available Node.js APIs.

The Edge Runtime is ideal if you need to deliver dynamic, personalized content at low latency with small, simple functions. The Edge Runtime's speed comes from its minimal use of resources, but that can be limiting in many scenarios.

For example, code executed in the Edge Runtime [on Vercel cannot exceed between 1 MB and 4 MB ↗](#), this limit includes imported packages, fonts and files, and will vary depending on your deployment infrastructure.

Node.js Runtime

Using the Node.js runtime gives you access to all Node.js APIs, and all npm packages that rely on them. However, it's not as fast to start up as routes using the Edge runtime.

Deploying your Next.js application to a Node.js server will require managing, scaling, and configuring your infrastructure. Alternatively, you can consider deploying your Next.js application to a serverless platform like Vercel, which will handle this for you.

Serverless Node.js

Serverless is ideal if you need a scalable solution that can handle more complex computational loads than the Edge Runtime. With Serverless Functions on Vercel, for example, your overall code size is [50MB ↗](#) including imported packages, fonts, and files.

The downside compared to routes using the [Edge ↗](#) is that it can take hundreds of milliseconds for Serverless Functions to boot up before they begin processing requests. Depending on the amount of traffic your site receives, this could be a frequent occurrence as the functions are not frequently "warm".

> Menu

Pages Router > ... > Rendering > Incremental Static Regeneration (ISR)

Incremental Static Regeneration (ISR)

► Examples

Next.js allows you to create or update static pages *after* you've built your site. Incremental Static Regeneration (ISR) enables you to use static-generation on a per-page basis, **without needing to rebuild the entire site**. With ISR, you can retain the benefits of static while scaling to millions of pages.

Note: The `edge runtime` is currently not compatible with ISR, although you can leverage `stale-while-revalidate` by setting the `cache-control` header manually.

To use ISR, add the `revalidate` prop to `getStaticProps`:

```
1  function Blog({ posts }) {
2    return (
3      <ul>
4        {posts.map((post) => (
5          <li key={post.id}>{post.title}</li>
6        )))
7      </ul>
8    );
9  }
10
11 // This function gets called at build time on server-side.
12 // It may be called again, on a serverless function, if
13 // revalidation is enabled and a new request comes in
14 export async function getStaticProps() {
15   const res = await fetch('https://.../posts');
16   const posts = await res.json();
17
18   return {
19     props: {
20       posts,
21     },
22     // Next.js will attempt to re-generate the page:
23     // - When a request comes in
24     // - At most once every 10 seconds
25     revalidate: 10, // In seconds
26   };
}
```

```

27 }
28
29 // This function gets called at build time on server-side.
30 // It may be called again, on a serverless function, if
31 // the path has not been generated.
32 export async function getStaticPaths() {
33   const res = await fetch('https://.../posts');
34   const posts = await res.json();
35
36   // Get the paths we want to pre-render based on posts
37   const paths = posts.map((post) => ({
38     params: { id: post.id },
39   }));
40
41   // We'll pre-render only these paths at build time.
42   // { fallback: 'blocking' } will server-render pages
43   // on-demand if the path doesn't exist.
44   return { paths, fallback: 'blocking' };
45 }
46
47 export default Blog;

```

When a request is made to a page that was pre-rendered at build time, it will initially show the cached page.

- Any requests to the page after the initial request and before 10 seconds are also cached and instantaneous.
- After the 10-second window, the next request will still show the cached (stale) page
- Next.js triggers a regeneration of the page in the background.
- Once the page generates successfully, Next.js will invalidate the cache and show the updated page. If the background regeneration fails, the old page would still be unaltered.

When a request is made to a path that hasn't been generated, Next.js will server-render the page on the first request. Future requests will serve the static file from the cache. ISR on Vercel [persists the cache globally and handles rollbacks ↗](#).

Note: Check if your upstream data provider has caching enabled by default. You might need to disable (e.g. `useCdn: false`), otherwise a revalidation won't be able to pull fresh data to update the ISR cache. Caching can occur at a CDN (for an endpoint being requested) when it returns the `Cache-Control` header.

On-Demand Revalidation

If you set a `revalidate` time of `60`, all visitors will see the same generated version of your site for one minute. The only way to invalidate the cache is from someone visiting that page after the minute has

passed.

Starting with `v12.2.0`, Next.js supports On-Demand Incremental Static Regeneration to manually purge the Next.js cache for a specific page. This makes it easier to update your site when:

- Content from your headless CMS is created or updated
- Ecommerce metadata changes (price, description, category, reviews, etc.)

Inside `getStaticProps`, you do not need to specify `revalidate` to use on-demand revalidation. If `revalidate` is omitted, Next.js will use the default value of `false` (no revalidation) and only revalidate the page on-demand when `revalidate()` is called.

Note: [Middleware](#) won't be executed for On-Demand ISR requests. Instead, call `revalidate()` on the exact path that you want revalidated. For example, if you have `pages/blog/[slug].js` and a rewrite from `/post-1 -> /blog/post-1`, you would need to call `res.revalidate('/blog/post-1')`.

Using On-Demand Revalidation

First, create a secret token only known by your Next.js app. This secret will be used to prevent unauthorized access to the revalidation API Route. You can access the route (either manually or with a webhook) with the following URL structure:

>_ Terminal

`https://<your-site.com>/api/revalidate?secret=<token>`

Next, add the secret as an [Environment Variable](#) to your application. Finally, create the revalidation API Route:

js pages/api/revalidate.js

```
1 export default async function handler(req, res) {
2   // Check for secret to confirm this is a valid request
3   if (req.query.secret !== process.env.MY_SECRET_TOKEN) {
4     return res.status(401).json({ message: 'Invalid token' });
5   }
6
7   try {
8     // this should be the actual path not a rewritten path
9     // e.g. for "/blog/[slug]" this should be "/blog/post-1"
10    await res.revalidate('/path-to-revalidate');
11    return res.json({ revalidated: true });
12  } catch (err) {
13    // If there was an error, Next.js will continue
```

```
14     // to show the last successfully generated page
15     return res.status(500).send('Error revalidating');
16   }
17 }
```

[View our demo ↗](#) to see on-demand revalidation in action and provide feedback.

Testing on-Demand ISR during development

When running locally with `next dev`, `getStaticProps` is invoked on every request. To verify your on-demand ISR configuration is correct, you will need to create a [production build](#) and start the [production server](#):

>_ Terminal

```
1 $ next build
2 $ next start
```

Then, you can confirm that static pages have successfully revalidated.

Error handling and revalidation

If there is an error inside `getStaticProps` when handling background regeneration, or you manually throw an error, the last successfully generated page will continue to show. On the next subsequent request, Next.js will retry calling `getStaticProps`.

```
1 export async function getStaticProps() {
2   // If this request throws an uncaught error, Next.js will
3   // not invalidate the currently shown page and
4   // retry getStaticProps on the next request.
5   const res = await fetch('https://.../posts');
6   const posts = await res.json();
7
8   if (!res.ok) {
9     // If there is a server error, you might want to
10    // throw an error instead of returning so that the cache is not updated
11    // until the next successful request.
12    throw new Error(`Failed to fetch posts, received status ${res.status}`);
13  }
14
15  // If the request was successful, return the posts
16  // and revalidate every 10 seconds.
17  return {
18    props: {
```

```
19     posts,
20   },
21   revalidate: 10,
22 };
23 }
```

Self-hosting ISR

Incremental Static Regeneration (ISR) works on [self-hosted Next.js sites](#) out of the box when you use `next start`.

You can use this approach when deploying to container orchestrators such as [Kubernetes ↗](#) or [HashiCorp Nomad ↗](#). By default, generated assets will be stored in-memory on each pod. This means that each pod will have its own copy of the static files. Stale data may be shown until that specific pod is hit by a request.

To ensure consistency across all pods, you can disable in-memory caching. This will inform the Next.js server to only leverage assets generated by ISR in the file system.

You can use a shared network mount in your Kubernetes pods (or similar setup) to reuse the same file-system cache between different containers. By sharing the same mount, the `.next` folder which contains the `next/image` cache will also be shared and re-used.

To disable in-memory caching, set `isrMemoryCacheSize` to `0` in your `next.config.js` file:

JS next.config.js

```
1 module.exports = {
2   experimental: {
3     // Defaults to 50MB
4     isrMemoryCacheSize: 0,
5   },
6 };
```

Note: You might need to consider a race condition between multiple pods trying to update the cache at the same time, depending on how your shared mount is configured.

Version History

Version	Changes
v12.2.0	On-Demand ISR is stable
v12.1.0	On-Demand ISR added (beta).
v12.0.0	Bot-aware ISR fallback ↗ added.
v9.5.0	Base Path added.

> Menu

Pages Router > ... > Rendering > Server-side Rendering (SSR)

Server-side Rendering (SSR)

Also referred to as "SSR" or "Dynamic Rendering".

If a page uses **Server-side Rendering**, the page HTML is generated on **each request**.

To use Server-side Rendering for a page, you need to `export` an `async` function called `getServerSideProps`. This function will be called by the server on every request.

For example, suppose that your page needs to pre-render frequently updated data (fetched from an external API). You can write `getServerSideProps` which fetches this data and passes it to `Page` like below:

```
1 export default function Page({ data }) {
2   // Render data...
3 }
4
5 // This gets called on every request
6 export async function getServerSideProps() {
7   // Fetch data from external API
8   const res = await fetch(`https://.../data`);
9   const data = await res.json();
10
11 // Pass data to the page via props
12 return { props: { data } };
13 }
```

As you can see, `getServerSideProps` is similar to `getStaticProps`, but the difference is that `getServerSideProps` is run on every request instead of on build time.

To learn more about how `getServerSideProps` works, check out our [Data Fetching documentation](#).

> Menu

Pages Router > ... > Rendering > Static Site Generation (SSG)

Static Site Generation (SSG)

► Examples

If a page uses **Static Generation**, the page HTML is generated at **build time**. That means in production, the page HTML is generated when you run `next build`. This HTML will then be reused on each request. It can be cached by a CDN.

In Next.js, you can statically generate pages **with or without data**. Let's take a look at each case.

Static Generation without data

By default, Next.js pre-renders pages using Static Generation without fetching data. Here's an example:

```
1 function About() {  
2   return <div>About</div>;  
3 }  
4  
5 export default About;
```

Note that this page does not need to fetch any external data to be pre-rendered. In cases like this, Next.js generates a single HTML file per page during build time.

Static Generation with data

Some pages require fetching external data for pre-rendering. There are two scenarios, and one or both might apply. In each case, you can use these functions that Next.js provides:

1. Your page **content** depends on external data: Use `getStaticProps`.
2. Your page **paths** depend on external data: Use `getStaticPaths` (usually in addition to `getStaticProps`).

Scenario 1: Your page content depends on external data

Example: Your blog page might need to fetch the list of blog posts from a CMS (content management system).

```
1 // TODO: Need to fetch `posts` (by calling some API endpoint)
2 //       before this page can be pre-rendered.
3 export default function Blog({ posts }) {
4   return (
5     <ul>
6       {posts.map((post) => (
7         <li>{post.title}</li>
8       )))
9     </ul>
10  );
11}
```

To fetch this data on pre-render, Next.js allows you to `export` an `async` function called `getStaticProps` from the same file. This function gets called at build time and lets you pass fetched data to the page's `props` on pre-render.

```
1 export default function Blog({ posts }) {
2   // Render posts...
3 }
4
5 // This function gets called at build time
6 export async function getStaticProps() {
7   // Call an external API endpoint to get posts
8   const res = await fetch('https://.../posts');
9   const posts = await res.json();
10
11  // By returning { props: { posts } }, the Blog component
12  // will receive `posts` as a prop at build time
13  return {
14    props: {
15      posts,
16    },
17  };
18}
```

To learn more about how `getStaticProps` works, check out the [Data Fetching documentation](#).

Scenario 2: Your page paths depend on external data

Next.js allows you to create pages with **dynamic routes**. For example, you can create a file called `pages/posts/[id].js` to show a single blog post based on `id`. This will allow you to show a blog post with `id: 1` when you access `posts/1`.

To learn more about dynamic routing, check the [Dynamic Routing documentation](#).

However, which `id` you want to pre-render at build time might depend on external data.

Example: suppose that you've only added one blog post (with `id: 1`) to the database. In this case, you'd only want to pre-render `posts/1` at build time.

Later, you might add the second post with `id: 2`. Then you'd want to pre-render `posts/2` as well.

So your page **paths** that are pre-rendered depend on external data**. To handle this, Next.js lets you `export` an `async` function called `getStaticPaths` from a dynamic page (`pages/posts/[id].js` in this case). This function gets called at build time and lets you specify which paths you want to pre-render.

```
1 // This function gets called at build time
2 export async function getStaticPaths() {
3     // Call an external API endpoint to get posts
4     const res = await fetch('https://.../posts');
5     const posts = await res.json();
6
7     // Get the paths we want to pre-render based on posts
8     const paths = posts.map((post) => ({
9         params: { id: post.id },
10    }));
11
12     // We'll pre-render only these paths at build time.
13     // { fallback: false } means other routes should 404.
14     return { paths, fallback: false };
15 }
```

Also in `pages/posts/[id].js`, you need to export `getStaticProps` so that you can fetch the data about the post with this `id` and use it to pre-render the page:

```
1 export default function Post({ post }) {
2     // Render post...
3 }
4
5 export async function getStaticPaths() {
6     // ...
7 }
8
9 // This also gets called at build time
10 export async function getStaticProps({ params }) {
11     // params contains the post `id`.
12     // If the route is like /posts/1, then params.id is 1
13     const res = await fetch(`https://.../posts/${params.id}`);
14     const post = await res.json();
15
16     // Pass post data to the page via props
17     return { props: { post } };
18 }
```

When should I use Static Generation?

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 and portfolios
- 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 cases like this, you can do one of the following:

- Use Static Generation with **Client-side data fetching**: You can skip pre-rendering some parts of a page and then use client-side JavaScript to populate them. To learn more about this approach, check out the [Data Fetching documentation](#).
- Use **Server-Side Rendering**: Next.js pre-renders a page on each request. It will be slower because the page cannot be cached by a CDN, but the pre-rendered page will always be up-to-date. We'll talk about this approach below.

> Menu

Pages Router > Building Your Application > Routing

Routing

The Pages Router has a file-system based router built on concepts of pages. When a file is added to the `pages` directory it's automatically available as a route. Learn more about routing in the Pages Router:

Pages and Layouts

Create your first page and shared layout with the Pages Router.

Dynamic Routes

Dynamic Routes are pages that allow you to add custom params to your URLs. Start creating Dynamic Routes and learn more here.

Linking and Navigating

Learn how navigation works in Next.js, and how to use the Link Component and `useRouter` hook.

Custom App

Control page initialization and add a layout that persists for all pages by overriding the default App component used by Next.js.

Custom Document

Extend the default document markup added by Next.js.

Custom Errors

Override and extend the built-in Error page to handle custom errors.

API Routes

Next.js supports API Routes, which allow you to build your API without leaving your Next.js app. Learn how it works here.

Internationalization

Next.js has built-in support for internationalized routing and language detection. Learn more here.

Authenticating

Learn about authentication patterns in Next.js apps and explore a few examples.

Middleware

Learn how to use Middleware to run code before a request is completed.

> Menu

Pages Router > ... > Routing > API Routes

API Routes

► Examples

Note: If you are using the App Router, you can use [Server Components](#) or [Route Handlers](#) instead of API Routes.

API routes provide a solution to build your **API** with Next.js.

Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a `page`. They are server-side only bundles and won't increase your client-side bundle size.

For example, the following API route `pages/api/user.js` returns a `json` response with a status code of `200`:

```
1 export default function handler(req, res) {  
2   res.status(200).json({ name: 'John Doe' });  
3 }
```

Note: API Routes will be affected by [pageExtensions](#) configuration in `next.config.js`.

For an API route to work, you need to export a function as default (a.k.a **request handler**), which then receives the following parameters:

- `req` : An instance of [http.IncomingMessage](#), plus some [pre-built middlewares](#)
- `res` : An instance of [http.ServerResponse](#), plus some [helper functions](#)

To handle different HTTP methods in an API route, you can use `req.method` in your request handler, like so:

```
1 export default function handler(req, res) {  
2   if (req.method === 'POST') {  
3     // Process a POST request  
4   } else {
```

```
5      // Handle any other HTTP method
6  }
7 }
```

To fetch API endpoints, take a look into any of the examples at the start of this section.

Use Cases

For new projects, you can build your entire API with API Routes. If you have an existing API, you do not need to forward calls to the API through an API Route. Some other use cases for API Routes are:

- Masking the URL of an external service (e.g. `/api/secret` instead of `https://company.com/secret-url`)
 - Using [Environment Variables](#) on the server to securely access external services.
-

Caveats

- API Routes [do not specify CORS headers ↗](#), meaning they are **same-origin only** by default. You can customize such behavior by wrapping the request handler with the [CORS request helpers ↗](#).
 - API Routes can't be used with `output: 'export'`
-

Request Helpers

API Routes provide built-in request helpers which parse the incoming request (`req`):

- `req.cookies` - An object containing the cookies sent by the request. Defaults to `{}`
- `req.query` - An object containing the [query string ↗](#). Defaults to `{}`
- `req.body` - An object containing the body parsed by `content-type`, or `null` if no body was sent

Custom config

Every API Route can export a `config` object to change the default configuration, which is the following:

```
1 export const config = {
2   api: {
3     bodyParser: {
4       sizeLimit: '1mb',
5     },
6   },
7 };
```

The `api` object includes all config options available for API Routes.

`bodyParser` is automatically enabled. If you want to consume the body as a `Stream` or with [raw-body](#) ↗, you can set this to `false`.

One use case for disabling the automatic `bodyParsing` is to allow you to verify the raw body of a [webhook](#) request, for example [from GitHub](#) ↗.

```
1 export const config = {
2   api: {
3     bodyParser: false,
4   },
5 };
```

`bodyParser.sizeLimit` is the maximum size allowed for the parsed body, in any format supported by [bytes](#) ↗, like so:

```
1 export const config = {
2   api: {
3     bodyParser: {
4       sizeLimit: '500kb',
5     },
6   },
7 };
```

`externalResolver` is an explicit flag that tells the server that this route is being handled by an external resolver like `express` or `connect`. Enabling this option disables warnings for unresolved requests.

```
1 export const config = {
2   api: {
3     externalResolver: true,
4   },
5 };
```

`responseLimit` is automatically enabled, warning when an API Routes' response body is over 4MB.

If you are not using Next.js in a serverless environment, and understand the performance implications of not using a CDN or dedicated media host, you can set this limit to `false`.

```
1 export const config = {  
2   api: {  
3     responseLimit: false,  
4   },  
5 };
```

`responseLimit` can also take the number of bytes or any string format supported by `bytes`, for example `1000`, `'500kb'` or `'3mb'`. This value will be the maximum response size before a warning is displayed. Default is 4MB. (see above)

```
1 export const config = {  
2   api: {  
3     responseLimit: '8mb',  
4   },  
5 };
```

Extending the `req` / `res` objects with TypeScript

For better type-safety, it is not recommended to extend the `req` and `res` objects. Instead, use functions to work with them:

ts utils/cookies.ts

```
1 import { serialize, CookieSerializeOptions } from 'cookie';
2 import { NextApiResponse } from 'next';
3
4 /**
5  * This sets `cookie` using the `res` object
6 */
7 export const setCookie = (
8   res: NextApiResponse,
9   name: string,
10  value: unknown,
11  options: CookieSerializeOptions = {},
12 ) => {
13   const stringValue =
14     typeof value === 'object' ? `j:${JSON.stringify(value)}` : String(value);
15
16   if (typeof options.maxAge === 'number') {
17     options.expires = new Date(Date.now() + options.maxAge * 1000);
18   }
19
20   res.setHeader('Set-Cookie', serialize(name, stringValue, options));
21};
```

TS pages/api/cookies.ts

```
1 import { NextApiRequest, NextApiResponse } from 'next';
2 import { setCookie } from '../utils/cookies';
3
4 const handler = (req: NextApiRequest, res: NextApiResponse) => {
5   // Calling our pure function using the `res` object, it will add the `set-cookie` header
6   // Add the `set-cookie` header on the main domain and expire after 30 days
7   setCookie(res, 'Next.js', 'api-middleware!', { path: '/', maxAge: 2592000 });
8   // Return the `set-cookie` header so we can display it in the browser and show that it
9   res.end(res.getHeader('Set-Cookie'));
10 };
11
12 export default handler;
```

If you can't avoid these objects from being extended, you have to create your own type to include the extra properties:

TS pages/api/foo.ts

```
1 import { NextApiRequest, NextApiResponse } from 'next';
2 import { withFoo } from 'external-lib-foo';
3
4 type NextApiRequestWithFoo = NextApiRequest & {
5   foo: (bar: string) => void;
6 };
7
8 const handler = (req: NextApiRequestWithFoo, res: NextApiResponse) => {
```

```
9   req.foo('bar'); // we can now use `req.foo` without type errors
10  res.end('ok');
11 }
12
13 export default withFoo(handler);
```

Keep in mind this is not safe since the code will still compile even if you remove `withFoo()` from the export.

Response Helpers

The [Server Response object ↗](#), (often abbreviated as `res`) includes a set of Express.js-like helper methods to improve the developer experience and increase the speed of creating new API endpoints.

The included helpers are:

- `res.status(code)` - A function to set the status code. `code` must be a valid [HTTP status code ↗](#)
- `res.json(body)` - Sends a JSON response. `body` must be a [serializable object ↗](#)
- `res.send(body)` - Sends the HTTP response. `body` can be a `string`, an `object` or a `Buffer`
- `res.redirect([status,] path)` - Redirects to a specified path or URL. `status` must be a valid [HTTP status code ↗](#). If not specified, `status` defaults to "307" "Temporary redirect".
- `res.revalidate(urlPath)` - [Revalidate a page on demand](#) using `getStaticProps`. `urlPath` must be a `string`.

Setting the status code of a response

When sending a response back to the client, you can set the status code of the response.

The following example sets the status code of the response to `200` (`OK`) and returns a `message` property with the value of `Hello from Next.js!` as a JSON response:

```
1 export default function handler(req, res) {
2   res.status(200).json({ message: 'Hello from Next.js!' });
3 }
```

Sending a JSON response

When sending a response back to the client you can send a JSON response, this must be a [serializable object ↗](#). In a real world application you might want to let the client know the status of the request

depending on the result of the requested endpoint.

The following example sends a JSON response with the status code `200` (OK) and the result of the async operation. It's contained in a try catch block to handle any errors that may occur, with the appropriate status code and error message caught and sent back to the client:

```
1 export default async function handler(req, res) {  
2   try {  
3     const result = await someAsyncOperation();  
4     res.status(200).json({ result });  
5   } catch (err) {  
6     res.status(500).json({ error: 'failed to load data' });  
7   }  
8 }
```

Sending a HTTP response

Sending an HTTP response works the same way as when sending a JSON response. The only difference is that the response body can be a `string`, an `object` or a `Buffer`.

The following example sends a HTTP response with the status code `200` (OK) and the result of the async operation.

```
1 export default async function handler(req, res) {  
2   try {  
3     const result = await someAsyncOperation();  
4     res.status(200).send({ result });  
5   } catch (err) {  
6     res.status(500).send({ error: 'failed to fetch data' });  
7   }  
8 }
```

Redirects to a specified path or URL

Taking a form as an example, you may want to redirect your client to a specified path or URL once they have submitted the form.

The following example redirects the client to the `/` path if the form is successfully submitted:

```
1 export default async function handler(req, res) {  
2   const { name, message } = req.body;  
3   try {  
4     await handleFormInputAsync({ name, message });  
5     res.redirect(307, '/');  
6   } catch (err) {
```

```
7     res.status(500).send({ error: 'failed to fetch data' });
8   }
9 }
```

Adding TypeScript types

You can make your response handlers more type-safe by importing the `NextApiRequest` and `NextApiResponse` types from `next`, in addition to those, you can also type your response data:

```
1 import type { NextApiRequest, NextApiResponse } from 'next';
2
3 type ResponseData = {
4   message: string;
5 };
6
7 export default function handler(
8   req: NextApiRequest,
9   res: NextApiResponse<ResponseData>,
10 ) {
11   res.status(200).json({ message: 'Hello from Next.js!' });
12 }
```

Note: The body of `NextApiRequest` is `any` because the client may include any payload. You should validate the type/shape of the body at runtime before using it.

Dynamic API Routes

API routes support [dynamic routes](#), and follow the same file naming rules used for `pages`.

For example, the API route `pages/api/post/[pid].js` has the following code:

```
1 export default function handler(req, res) {
2   const { pid } = req.query;
3   res.end(`Post: ${pid}`);
4 }
```

Now, a request to `/api/post/abc` will respond with the text: `Post: abc`.

Index routes and Dynamic API routes

A very common RESTful pattern is to set up routes like this:

- GET `api/posts` - gets a list of posts, probably paginated
- GET `api/posts/12345` - gets post id 12345

We can model this in two ways:

- Option 1:
 - `/api/posts.js`
 - `/api/posts/[postId].js`
- Option 2:
 - `/api/posts/index.js`
 - `/api/posts/[postId].js`

Both are equivalent. A third option of only using `/api/posts/[postId].js` is not valid because Dynamic Routes (including Catch-all routes - see below) do not have an `undefined` state and `GET api/posts` will not match `/api/posts/[postId].js` under any circumstances.

Catch all API routes

API Routes can be extended to catch all paths by adding three dots (`...`) inside the brackets. For example:

- `pages/api/post/[...slug].js` matches `/api/post/a`, but also `/api/post/a/b`, `/api/post/a/b/c` and so on.

Note: You can use names other than `slug`, such as: `[...param]`

Matched parameters will be sent as a query parameter (`slug` in the example) to the page, and it will always be an array, so, the path `/api/post/a` will have the following `query` object:

```
{ "slug": [ "a" ] }
```

And in the case of `/api/post/a/b`, and any other matching path, new parameters will be added to the array, like so:

```
{ "slug": [ "a", "b" ] }
```

An API route for `pages/api/post/[...slug].js` could look like this:

```
1 export default function handler(req, res) {
```

```
2   const { slug } = req.query;
3   res.end(`Post: ${slug.join(', ')}`);
4 }
```

Now, a request to `/api/post/a/b/c` will respond with the text: `Post: a, b, c`.

Optional catch all API routes

Catch all routes can be made optional by including the parameter in double brackets (`[[...slug]]`).

For example, `pages/api/post/[[...slug]].js` will match `/api/post`, `/api/post/a`, `/api/post/a/b`, and so on.

The main difference between catch all and optional catch all routes is that with optional, the route without the parameter is also matched (`/api/post` in the example above).

The `query` objects are as follows:

```
1 { } // GET `/api/post` (empty object)
2 { "slug": ["a"] } // `GET /api/post/a` (single-element array)
3 { "slug": ["a", "b"] } // `GET /api/post/a/b` (multi-element array)
```

Caveats

- Predefined API routes take precedence over dynamic API routes, and dynamic API routes over catch all API routes. Take a look at the following examples:
 - `pages/api/post/create.js` - Will match `/api/post/create`
 - `pages/api/post/[pid].js` - Will match `/api/post/1`, `/api/post/abc`, etc. But not `/api/post/create`
 - `pages/api/post/[[...slug]].js` - Will match `/api/post/1/2`, `/api/post/a/b/c`, etc. But not `/api/post/create`, `/api/post/abc`

Edge API Routes

Edge API Routes enable you to build high performance APIs with Next.js. Using the [Edge Runtime](#), they are often faster than Node.js-based API Routes. This performance improvement does come with [constraints](#), like not having access to native Node.js APIs. Instead, Edge API Routes are built on standard Web APIs.

Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a page. They are server-side only bundles and won't increase your client-side bundle size.

Examples

Basic

```
1 export const config = {
2   runtime: 'edge',
3 };
4
5 export default (req) => new Response('Hello world!');
```

JSON Response

```
1 import type { NextRequest } from 'next/server';
2
3 export const config = {
4   runtime: 'edge',
5 };
6
7 export default async function handler(req: NextRequest) {
8   return new Response(
9     JSON.stringify({
10       name: 'Jim Halpert',
11     }),
12     {
13       status: 200,
14       headers: {
15         'content-type': 'application/json',
16       },
17     },
18   );
19 }
```

Cache-Control

```
1 import type { NextRequest } from 'next/server';
2
3 export const config = {
4   runtime: 'edge',
5 };
6
7 export default async function handler(req: NextRequest) {
8   return new Response(
9     JSON.stringify({
10       name: 'Jim Halpert',
11     }),
12     {
13       cacheControl: 'no-store',
14     },
15   );
16 }
```

```
13     status: 200,
14     headers: {
15       'content-type': 'application/json',
16       'cache-control': 'public, s-maxage=1200, stale-while-revalidate=600',
17     },
18   },
19 );
20 }
```

Query Parameters

```
1 import type { NextRequest } from 'next/server';
2
3 export const config = {
4   runtime: 'edge',
5 };
6
7 export default async function handler(req: NextRequest) {
8   const { searchParams } = new URL(req.url);
9   const email = searchParams.get('email');
10  return new Response(email);
11 }
```

Forwarding Headers

```
1 import { type NextRequest } from 'next/server';
2
3 export const config = {
4   runtime: 'edge',
5 };
6
7 export default async function handler(req: NextRequest) {
8   const authorization = req.cookies.get('authorization')?.value;
9   return fetch('https://backend-api.com/api/protected', {
10     method: req.method,
11     headers: {
12       authorization,
13     },
14     redirect: 'manual',
15   });
16 }
```

Configuring Regions (for deploying)

You may want to restrict your edge function to specific regions when deploying so that you can colocate near your data sources ensuring lower response times which can be achieved as shown.

Note: This configuration is available in v12.3.2 of Next.js and up.

```
1 import { NextResponse } from 'next/server';
2
3 export const config = {
4   regions: ['sfo1', 'iad1'], // defaults to 'all'
5 };
6
7 export default async function handler(req: NextRequest) {
8   const myData = await getNearbyData();
9   return NextResponse.json(myData);
10 }
```

Differences between API Routes

Edge API Routes use the [Edge Runtime](#), whereas API Routes use the [Node.js runtime](#).

Edge API Routes can [stream responses](#) from the server and run *after* cached files (e.g. HTML, CSS, JavaScript) have been accessed. Server-side streaming can help improve performance with faster [Time To First Byte \(TTFB\)](#) ↗.

Note: Using [Edge Runtime](#) with `getServerSideProps` does not give you access to the response object. If you need access to `res`, you should use the [Node.js runtime](#) by setting `runtime: 'nodejs'`.

View the [supported APIs](#) and [unsupported APIs](#) for the Edge Runtime.

> Menu

Pages Router > ... > Routing > Authenticating

Authenticating

Authentication verifies who a user is, while authorization controls what a user can access. Next.js supports multiple authentication patterns, each designed for different use cases. This page will go through each case so that you can choose based on your constraints.

Authentication Patterns

The first step to identifying which authentication pattern you need is understanding the [data-fetching strategy](#) you want. We can then determine which authentication providers support this strategy. There are two main patterns:

- Use [static generation](#) to server-render a loading state, followed by fetching user data client-side.
- Fetch user data [server-side](#) to eliminate a flash of unauthenticated content.

Authenticating Statically Generated Pages

Next.js automatically determines that a page is static if there are no blocking data requirements. This means the absence of `getServerSideProps` and `getInitialProps` in the page. Instead, your page can render a loading state from the server, followed by fetching the user client-side.

One advantage of this pattern is it allows pages to be served from a global CDN and preloaded using [next/link](#). In practice, this results in a faster TTI ([Time to Interactive ↗](#)).

Let's look at an example for a profile page. This will initially render a loading skeleton. Once the request for a user has finished, it will show the user's name:

JS pages/profile.js



```
1 import useUser from '../lib/useUser';
2 import Layout from '../components/Layout';
```

```

3
4 const Profile = () => {
5   // Fetch the user client-side
6   const { user } = useUser({ redirectTo: '/login' });
7
8   // Server-render loading state
9   if (!user || user.isLoggedIn === false) {
10     return <Layout>Loading...</Layout>;
11   }
12
13   // Once the user request finishes, show the user
14   return (
15     <Layout>
16       <h1>Your Profile</h1>
17       <pre>{JSON.stringify(user, null, 2)}</pre>
18     </Layout>
19   );
20 };
21
22 export default Profile;

```

You can view this [example in action ↗](#). Check out the [with-iron-session ↗](#) example to see how it works.

Authenticating Server-Rendered Pages

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`.

```

1 export async function getServerSideProps(context) {
2   return {
3     props: {}, // Will be passed to the page component as props
4   };
5 }

```

Let's transform the profile example to use [server-side rendering](#). If there's a session, return `user` as a prop to the `Profile` component in the page. Notice there is not a loading skeleton in [this example ↗](#).

js pages/profile.js



```

1 import withSession from '../lib/session';
2 import Layout from '../components/Layout';
3
4 export const getServerSideProps = withSession(async function ({ req, res }) {
5   const { user } = req.session;
6
7   if (!user) {
8     return {
9       redirect: {
10         destination: '/login',

```

```

11         permanent: false,
12     },
13 }
14 }
15
16 return {
17     props: { user },
18 };
19 });
20
21 const Profile = ({ user }) => {
22 // Show the user. No loading state is required
23 return (
24     <Layout>
25         <h1>Your Profile</h1>
26         <pre>{JSON.stringify(user, null, 2)}</pre>
27     </Layout>
28 );
29 };
30
31 export default Profile;

```

An advantage of this pattern is preventing a flash of unauthenticated content before redirecting. It's important to note fetching user data in `getServerSideProps` will block rendering until the request to your authentication provider resolves. To prevent creating a bottleneck and increasing your TTFB ([Time to First Byte ↗](#)), you should ensure your authentication lookup is fast. Otherwise, consider [static generation](#).

Authentication Providers

Now that we've discussed authentication patterns, let's look at specific providers and explore how they're used with Next.js.

Bring Your Own Database

▼ Examples

- [with-iron-session ↗](#)
- [next-auth-example ↗](#)

If you have an existing database with user data, you'll likely want to utilize an open-source solution that's provider agnostic.

- If you want a low-level, encrypted, and stateless session utility use [iron-session ↗](#).

- If you want a full-featured authentication system with built-in providers (Google, Facebook, GitHub...), JWT, JWE, email/password, magic links and more... use [next-auth ↗](#).

Both of these libraries support either authentication pattern. If you're interested in [Passport ↗](#), we also have examples for it using secure and encrypted cookies:

- [with-passport ↗](#)
- [with-passport-and-next-connect ↗](#)

Other Providers

To see examples with other authentication providers, check out the [examples folder ↗](#).

▼ Examples

- [Auth0 ↗](#)
- [Clerk ↗](#)
- [Firebase ↗](#)
- [Magic ↗](#)
- [Nhost ↗](#)
- [Ory ↗](#)
- [Supabase ↗](#)
- [Supertokens ↗](#)
- [Userbase ↗](#)

> Menu

Pages Router > ... > Routing > Custom App

Custom App

Next.js uses the `App` component to initialize pages. You can override it and control the page initialization and:

- Persist layouts between page changes
- Keeping state when navigating pages
- Inject additional data into pages
- [Add global CSS](#)

To override the default `App`, create the file `pages/_app.js` as shown below:

`JS pages/_app.js`

```
1 export default function MyApp({ Component, pageProps }) {  
2   return <Component {...pageProps} />;  
3 }
```

The `Component` prop is the active `page`, so whenever you navigate between routes, `Component` will change to the new `page`. Therefore, any props you send to `Component` will be received by the `page`.

`pageProps` is an object with the initial props that were preloaded for your page by one of our [data fetching methods](#), otherwise it's an empty object.

The `App.getInitialProps` receives a single argument called `context.ctx`. It's an object with the same set of properties as the `context object` in `getInitialProps`.

Caveats

- If your app is running and you added a custom `App`, you'll need to restart the development server. Only required if `pages/_app.js` didn't exist before.

- Adding a custom `getInitialProps` in your `App` will disable [Automatic Static Optimization](#) in pages without [Static Generation](#).
- When you add `getInitialProps` in your custom app, you must `import App from "next/app"`, call `App.getInitialProps(appContext)` inside `getInitialProps` and merge the returned object into the return value.
- `App` does not support Next.js [Data Fetching methods](#) like `getStaticProps` or `getServerSideProps`. If you need global data fetching, consider [incrementally adopting the `app/` directory](#).

TypeScript

If you're using TypeScript, take a look at [our TypeScript documentation](#).

> Menu

Pages Router > ... > Routing > Custom Document

Custom Document

A custom `Document` can update the `<html>` and `<body>` tags used to render a `Page`. This file is only rendered on the server, so event handlers like `onClick` cannot be used in `_document`.

To override the default `Document`, create the file `pages/_document.js` as shown below:

```
1 import { Html, Head, Main, NextScript } from 'next/document';
2
3 export default function Document() {
4   return (
5     <Html>
6       <Head />
7       <body>
8         <Main />
9         <NextScript />
10      </body>
11    </Html>
12  );
13}
```

The code above is the default `Document` added by Next.js. Custom attributes are allowed as props. For example, we might want to add `lang="en"` to the `<html>` tag:

```
<Html lang="en">
```

Or add a `className` to the `body` tag:

```
<body className="bg-white">
```

`<Html>`, `<Head />`, `<Main />` and `<NextScript />` are required for the page to be properly rendered.

Caveats

- The `<Head />` component used in `_document` is not the same as `next/head`. The `<Head />` component used here should only be used for any `<head>` code that is common for all pages. For all other cases, such as `<title>` tags, we recommend using `next/head` in your pages or components.
- React components outside of `<Main />` will not be initialized by the browser. Do *not* add application logic here or custom CSS (like `styled-jsx`). If you need shared components in all your pages (like a menu or a toolbar), read [Layouts](#) instead.
- `Document` currently does not support Next.js [Data Fetching methods](#) like `getStaticProps` or `getServerSideProps`.

Customizing `renderPage`

Note: This is advanced and only needed for libraries like CSS-in-JS to support server-side rendering. This is not needed for built-in `styled-jsx` support.

For [React 18](#) support, we recommend avoiding customizing `getInitialProps` and `renderPage`, if possible.

The `ctx` object shown below is equivalent to the one received in `getInitialProps`, with the addition of `renderPage`.

```
1 import Document, { Html, Head, Main, NextScript } from 'next/document';
2
3 class MyDocument extends Document {
4   static async getInitialProps(ctx) {
5     const originalRenderPage = ctx.renderPage;
6
7     // Run the React rendering logic synchronously
8     ctx.renderPage = () =>
9       originalRenderPage({
10         // Useful for wrapping the whole react tree
11         enhanceApp: (App) => App,
12         // Useful for wrapping in a per-page basis
13         enhanceComponent: (Component) => Component,
14       });
15
16     // Run the parent `getInitialProps`, it now includes the custom `renderPage`
17     const initialProps = await Document.getInitialProps(ctx);
18   }
}
```

```
19     return initialProps;
20 }
21
22 render() {
23     return (
24         <Html>
25             <Head />
26             <body>
27                 <Main />
28                 <NextScript />
29             </body>
30         </Html>
31     );
32 }
33 }
34
35 export default MyDocument;
```

Note: `getInitialProps` in `_document` is not called during client-side transitions.

TypeScript

You can use the built-in `DocumentContext` type and change the file name to `./pages/_document.tsx` like so:

```
1 import Document, { DocumentContext, DocumentInitialProps } from 'next/document';
2
3 class MyDocument extends Document {
4     static async getInitialProps(
5         ctx: DocumentContext,
6     ): Promise<DocumentInitialProps> {
7         const initialProps = await Document.getInitialProps(ctx);
8
9         return initialProps;
10    }
11 }
12
13 export default MyDocument;
```

> Menu

Pages Router > ... > Routing > Custom Errors

Custom Errors

404 Page

A 404 page may be accessed very often. Server-rendering an error page for every visit increases the load of the Next.js server. This can result in increased costs and slow experiences.

To avoid the above pitfalls, Next.js provides a static 404 page by default without having to add any additional files.

Customizing The 404 Page

To create a custom 404 page you can create a `pages/404.js` file. This file is statically generated at build time.

JS pages/404.js

```
1 export default function Custom404() {  
2   return <h1>404 - Page Not Found</h1>;  
3 }
```

Note: You can use `getStaticProps` inside this page if you need to fetch data at build time.

500 Page

Server-rendering an error page for every visit adds complexity to responding to errors. To help users get responses to errors as fast as possible, Next.js provides a static 500 page by default without having to add any additional files.

Customizing The 500 Page

To customize the 500 page you can create a `pages/500.js` file. This file is statically generated at build time.

JS pages/500.js

```
1 export default function Custom500() {
2   return <h1>500 - Server-side error occurred</h1>;
3 }
```

Note: You can use `getStaticProps` inside this page if you need to fetch data at build time.

More Advanced Error Page Customizing

500 errors are handled both client-side and server-side by the `Error` component. If you wish to override it, define the file `pages/_error.js` and add the following code:

```
1 function Error({ statusCode }) {
2   return (
3     <p>
4       {statusCode
5         ? `An error ${statusCode} occurred on server`
6         : 'An error occurred on client'}
7     </p>
8   );
9 }
10
11 Error.getInitialProps = ({ res, err }) => {
12   const statusCode = res ? res.statusCode : err ? err.statusCode : 404;
13   return { statusCode };
14 };
15
16 export default Error;
```

`pages/_error.js` is only used in production. In development you'll get an error with the call stack to know where the error originated from.

Reusing the built-in error page

If you want to render the built-in error page you can by importing the `Error` component:

```
1 import Error from 'next/error';
2
```

```
3 export async function getServerSideProps() {
4   const res = await fetch('https://api.github.com/repos/vercel/next.js');
5   const errorCode = res.ok ? false : res.status;
6   const json = await res.json();
7
8   return {
9     props: { errorCode, stars: json.stargazers_count },
10    };
11 }
12
13 export default function Page({ errorCode, stars }) {
14   if (errorCode) {
15     return <Error statusCode={errorCode} />;
16   }
17
18   return <div>Next stars: {stars}</div>;
19 }
```

The `Error` component also takes `title` as a property if you want to pass in a text message along with a `statusCode`.

If you have a custom `Error` component be sure to import that one instead. `next/error` exports the default component used by Next.js.

Caveats

- `Error` does not currently support Next.js Data Fetching methods like `getStaticProps` or `getServerSideProps`.
- `_error`, like `_app`, is a reserved pathname. `_error` is used to define the customized layouts and behaviors of the error pages. `/_error` will render 404 when accessed directly via `routing` or rendering in a `custom server`.

> Menu

Pages Router > ... > Routing > Dynamic Routes

Dynamic Routes

When you don't know the exact segment names ahead of time and want to create routes from dynamic data, you can use Dynamic Segments that are filled in at request time or [prerendered](#) at build time.

Convention

A Dynamic Segment can be created by wrapping a folder's name in square brackets: `[folderName]`. For example, `[id]` or `[slug]`.

Dynamic Segments can be accessed from [useRouter](#).

Example

For example, a blog could include the following route `pages/blog/[slug].js` where `[slug]` is the Dynamic Segment for blog posts.

```
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
<code>pages/blog/[slug].js</code>	<code>/blog/a</code>	<code>{ slug: 'a' }</code>
<code>pages/blog/[slug].js</code>	<code>/blog/b</code>	<code>{ slug: 'b' }</code>

Route	Example URL	params
pages/blog/[slug].js	/blog/c	{ slug: 'c' }

Catch-all Segments

Dynamic Segments can be extended to **catch-all** subsequent segments by adding an ellipsis inside the brackets `[...folderName]`.

For example, `pages/shop/[...slug].js` will match `/shop/clothes`, but also `/shop/clothes/tops`, `/shop/clothes/tops/t-shirts`, and so on.

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

Optional Catch-all Segments

Catch-all Segments can be made **optional** by including the parameter in double square brackets:

`[[...folderName]]`.

For example, `pages/shop/[[...slug]].js` will **also** match `/shop`, in addition to `/shop/clothes`, `/shop/clothes/tops`, `/shop/clothes/tops/t-shirts`.

The difference between **catch-all** and **optional catch-all** segments is that with optional, the route without the parameter is also matched (`/shop` in the example above).

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

Next Steps

For more information on what to do next, we recommend the following sections

Pages Router > ... > Routing

Linking and Navigating

Learn how navigation works in Next.js, and how to use the Link Component and `useRouter` hook.

Pages Router > ... > Functions

useRouter

Learn more about the API of the Next.js Router, and access the router instance in your page with the useRouter hook.

> Menu

Pages Router > ... > Routing > Internationalization

Internationalization (i18n) Routing

► Examples

Next.js has built-in support for internationalized ([i18n ↗](#)) routing since [v10.0.0](#). You can provide a list of locales, the default locale, and domain-specific locales and Next.js will automatically handle the routing.

The i18n routing support is currently meant to complement existing i18n library solutions like [react-intl ↗](#), [react-i18next ↗](#), [lingui ↗](#), [rosetta ↗](#), [next-intl ↗](#), [next-translate ↗](#), [next-multilingual ↗](#), [typescript-i18n ↗](#), and others by streamlining the routes and locale parsing.

Getting started

To get started, add the `i18n` config to your `next.config.js` file.

Locales are [UTS Locale Identifiers ↗](#), a standardized format for defining locales.

Generally a Locale Identifier is made up of a language, region, and script separated by a dash: `language-region-script`. The region and script are optional. An example:

- `en-US` - English as spoken in the United States
- `nl-NL` - Dutch as spoken in the Netherlands
- `nl` - Dutch, no specific region

If user locale is `nl-BE` and it is not listed in your configuration, they will be redirected to `nl` if available, or to the default locale otherwise. If you don't plan to support all regions of a country, it is therefore a good practice to include country locales that will act as fallbacks.

JS next.config.js

```
1 module.exports = {
```

```
2 i18n: {
3   // These are all the locales you want to support in
4   // your application
5   locales: ['en-US', 'fr', 'nl-NL'],
6   // This is the default locale you want to be used when visiting
7   // a non-locale prefixed path e.g. `/hello`
8   defaultLocale: 'en-US',
9   // This is a list of locale domains and the default locale they
10  // should handle (these are only required when setting up domain routing)
11  // Note: subdomains must be included in the domain value to be matched e.g. "fr.examp
12  domains: [
13    {
14      domain: 'example.com',
15      defaultLocale: 'en-US',
16    },
17    {
18      domain: 'example.nl',
19      defaultLocale: 'nl-NL',
20    },
21    {
22      domain: 'example.fr',
23      defaultLocale: 'fr',
24      // an optional http field can also be used to test
25      // locale domains locally with http instead of https
26      http: true,
27    },
28  ],
29 },
30 };
```

Locale Strategies

There are two locale handling strategies: Sub-path Routing and Domain Routing.

Sub-path Routing

Sub-path Routing puts the locale in the url path.

js next.config.js

```
1 module.exports = {
2   i18n: {
3     locales: ['en-US', 'fr', 'nl-NL'],
4     defaultLocale: 'en-US',
5   },
6 };
```

With the above configuration `en-US`, `fr`, and `nl-NL` will be available to be routed to, and `en-US` is the default locale. If you have a `pages/blog.js` the following urls would be available:

- `/blog`
- `/fr/blog`
- `/nl-nl/blog`

The default locale does not have a prefix.

Domain Routing

By using domain routing you can configure locales to be served from different domains:

`js` next.config.js

```
1 module.exports = {
2   i18n: {
3     locales: ['en-US', 'fr', 'nl-NL', 'nl-BE'],
4     defaultLocale: 'en-US',
5
6     domains: [
7       {
8         // Note: subdomains must be included in the domain value to be matched
9         // e.g. www.example.com should be used if that is the expected hostname
10        domain: 'example.com',
11        defaultLocale: 'en-US',
12      },
13      {
14        domain: 'example.fr',
15        defaultLocale: 'fr',
16      },
17      {
18        domain: 'example.nl',
19        defaultLocale: 'nl-NL',
20        // specify other locales that should be redirected
21        // to this domain
22        locales: ['nl-BE'],
23      },
24    ],
25  },
26};
```

For example if you have `pages/blog.js` the following urls will be available:

- `example.com/blog`
- `www.example.com/blog`
- `example.fr/blog`

- `example.nl/blog`
- `example.nl/nl-BE/blog`

Automatic Locale Detection

When a user visits the application root (generally `/`), Next.js will try to automatically detect which locale the user prefers based on the [Accept-Language](#) header and the current domain.

If a locale other than the default locale is detected, the user will be redirected to either:

- **When using Sub-path Routing:** The locale prefixed path
- **When using Domain Routing:** The domain with that locale specified as the default

When using Domain Routing, if a user with the `Accept-Language` header `fr;q=0.9` visits `example.com`, they will be redirected to `example.fr` since that domain handles the `fr` locale by default.

When using Sub-path Routing, the user would be redirected to `/fr`.

Prefixing the Default Locale

With Next.js 12 and [Middleware](#), we can add a prefix to the default locale with a [workaround](#).

For example, here's a `next.config.js` file with support for a few languages. Note the `"default"` locale has been added intentionally.

`JS` `next.config.js`

```
1 module.exports = {
2   i18n: {
3     locales: ['default', 'en', 'de', 'fr'],
4     defaultLocale: 'default',
5     localeDetection: false,
6   },
7   trailingSlash: true,
8 };
```

Next, we can use [Middleware](#) to add custom routing rules:

`TS` `middleware.ts`

```
1 import { NextRequest, NextResponse } from 'next/server';
```

```

2
3 const PUBLIC_FILE = /\.(\.*$)/;
4
5 export async function middleware(req: NextRequest) {
6   if (
7     req.nextUrl.pathname.startsWith('/_next') ||
8     req.nextUrl.pathname.includes('/api/') ||
9     PUBLIC_FILE.test(req.nextUrl.pathname)
10  ) {
11    return;
12  }
13
14  if (req.nextUrl.locale === 'default') {
15    const locale = req.cookies.get('NEXT_LOCALE')?.value || 'en';
16
17    return NextResponse.redirect(
18      new URL(
19        `/${locale}${req.nextUrl.pathname}${req.nextUrl.search}`,
20        req.url,
21      ),
22    );
23  }
24}

```

This [Middleware](#) skips adding the default prefix to [API Routes](#) and [public](#) files like fonts or images. If a request is made to the default locale, we redirect to our prefix `/en`.

Disabling Automatic Locale Detection

The automatic locale detection can be disabled with:

`next.config.js`

```

1 module.exports = {
2   i18n: {
3     localeDetection: false,
4   },
5 };

```

When `localeDetection` is set to `false` Next.js will no longer automatically redirect based on the user's preferred locale and will only provide locale information detected from either the locale based domain or locale path as described above.

Accessing the locale information

You can access the locale information via the Next.js router. For example, using the `useRouter()` hook the following properties are available:

- `locale` contains the currently active locale.
- `locales` contains all configured locales.
- `defaultLocale` contains the configured default locale.

When `pre-rendering` pages with `getStaticProps` or `getServerSideProps`, the locale information is provided in `the context` provided to the function.

When leveraging `getStaticPaths`, the configured locales are provided in the context parameter of the function under `locales` and the configured defaultLocale under `defaultLocale`.

Transition between locales

You can use `next/link` or `next/router` to transition between locales.

For `next/link`, a `locale` prop can be provided to transition to a different locale from the currently active one. If no `locale` prop is provided, the currently active `locale` is used during client-transitions. For example:

```
1 import Link from 'next/link';
2
3 export default function IndexPage(props) {
4   return (
5     <Link href="/another" locale="fr">
6       To /fr/another
7     </Link>
8   );
9 }
```

When using the `next/router` methods directly, you can specify the `locale` that should be used via the transition options. For example:

```
1 import { useRouter } from 'next/router';
2
3 export default function IndexPage(props) {
4   const router = useRouter();
5
6   return (
7     <div
8       onClick={() => {
```

```
9     router.push('/another', '/another', { locale: 'fr' });
10    }
11    >
12    to /fr/another
13  </div>
14 );
15 }
```

Note that to handle switching only the `locale` while preserving all routing information such as `dynamic route` query values or hidden href query values, you can provide the `href` parameter as an object:

```
1 import { useRouter } from 'next/router';
2 const router = useRouter();
3 const { pathname, asPath, query } = router;
4 // change just the locale and maintain all other route information including href's query
5 router.push({ pathname, query }, asPath, { locale: nextLocale });
```

See [here](#) for more information on the object structure for `router.push`.

If you have a `href` that already includes the locale you can opt-out of automatically handling the locale prefixing:

```
1 import Link from 'next/link';
2
3 export default function IndexPage(props) {
4   return (
5     <Link href="/fr/another" locale={false}>
6       To /fr/another
7     </Link>
8   );
9 }
```

Leveraging the `NEXT_LOCALE` cookie

Next.js supports overriding the accept-language header with a `NEXT_LOCALE=the-locale` cookie. This cookie can be set using a language switcher and then when a user comes back to the site it will leverage the locale specified in the cookie when redirecting from `/` to the correct locale location.

For example, if a user prefers the locale `fr` in their accept-language header but a `NEXT_LOCALE=en` cookie is set the `en` locale when visiting `/` the user will be redirected to the `en` locale location until the cookie is removed or expired.

Search Engine Optimization

Since Next.js knows what language the user is visiting it will automatically add the `lang` attribute to the `<html>` tag.

Next.js doesn't know about variants of a page so it's up to you to add the `hreflang` meta tags using `next/head`. You can learn more about `hreflang` in the [Google Webmasters documentation ↗](#).

How does this work with Static Generation?

Note that Internationalized Routing does not integrate with `output: 'export'` as it does not leverage the Next.js routing layer. Hybrid Next.js applications that do not use `output: 'export'` are fully supported.

Dynamic Routes and `getStaticProps` Pages

For pages using `getStaticProps` with [Dynamic Routes](#), all locale variants of the page desired to be prerendered need to be returned from `getStaticPaths`. Along with the `params` object returned for `paths`, you can also return a `locale` field specifying which locale you want to render. For example:

JS pages/blog/[slug].js

```
1 export const getStaticPaths = ({ locales }) => {
2   return {
3     paths: [
4       // if no `locale` is provided only the defaultLocale will be generated
5       { params: { slug: 'post-1' }, locale: 'en-US' },
6       { params: { slug: 'post-1' }, locale: 'fr' },
7     ],
8     fallback: true,
9   };
10 }
```

For [Automatically Statically Optimized](#) and non-dynamic `getStaticProps` pages, **a version of the page will be generated for each locale**. This is important to consider because it can increase build times depending on how many locales are configured inside `getStaticProps`.

For example, if you have 50 locales configured with 10 non-dynamic pages using `getStaticProps`, this means `getStaticProps` will be called 500 times. 50 versions of the 10 pages will be generated during each build.

To decrease the build time of dynamic pages with `getStaticProps`, use a [fallback mode](#). This allows you to return only the most popular paths and locales from `getStaticPaths` for prerendering during the build. Then, Next.js will build the remaining pages at runtime as they are requested.

Automatically Statically Optimized Pages

For pages that are [automatically statically optimized](#), a version of the page will be generated for each locale.

Non-dynamic `getStaticProps` Pages

For non-dynamic `getStaticProps` pages, a version is generated for each locale like above.

`getStaticProps` is called with each `locale` that is being rendered. If you would like to opt-out of a certain locale from being pre-rendered, you can return `notFound: true` from `getStaticProps` and this variant of the page will not be generated.

```
1 export async function getStaticProps({ locale }) {
2   // Call an external API endpoint to get posts.
3   // You can use any data fetching library
4   const res = await fetch(`https://.../posts?locale=${locale}`);
5   const posts = await res.json();
6
7   if (posts.length === 0) {
8     return {
9       notFound: true,
10    };
11  }
12
13  // By returning { props: posts }, the Blog component
14  // will receive `posts` as a prop at build time
15  return {
16    props: {
17      posts,
18    },
19  };
20 }
```

Limits for the i18n config

- `locales`: 100 total locales
- `domains`: 100 total locale domain items

Note: These limits have been added initially to prevent potential [performance issues at build time](#). You can workaround

these limits with custom routing using [Middleware](#) in Next.js 12.

> Menu

Pages Router > ... > Routing > Linking and Navigating

Linking and Navigating

The Next.js router allows you to do client-side route transitions between pages, similar to a single-page application.

A React component called `Link` is provided to do this client-side route transition.

```
1 import Link from 'next/link';
2
3 function Home() {
4   return (
5     <ul>
6       <li>
7         <Link href="/">Home</Link>
8       </li>
9       <li>
10        <Link href="/about">About Us</Link>
11      </li>
12      <li>
13        <Link href="/blog/hello-world">Blog Post</Link>
14      </li>
15    </ul>
16  );
17}
18
19 export default Home;
```

The example above uses multiple links. Each one maps a path (`href`) to a known page:

- `/` → `pages/index.js`
- `/about` → `pages/about.js`
- `/blog/hello-world` → `pages/blog/[slug].js`

Any `<Link />` in the viewport (initially or through scroll) will be prefetched by default (including the corresponding data) for pages using [Static Generation](#). The corresponding data for [server-rendered](#) routes is fetched *only when the `<Link />` is clicked*.

Linking to dynamic paths

You can also use interpolation to create the path, which comes in handy for [dynamic route segments](#). For example, to show a list of posts which have been passed to the component as a prop:

```
1 import Link from 'next/link';
2
3 function Posts({ posts }) {
4   return (
5     <ul>
6       {posts.map((post) => (
7         <li key={post.id}>
8           <Link href={`/blog/${encodeURIComponent(post.slug)}`}>
9             {post.title}
10            </Link>
11          </li>
12        )));
13      </ul>
14    );
15  }
16
17 export default Posts;
```

`encodeURIComponent` ↗ is used in the example to keep the path utf-8 compatible.

Alternatively, using a URL Object:

```
1 import Link from 'next/link';
2
3 function Posts({ posts }) {
4   return (
5     <ul>
6       {posts.map((post) => (
7         <li key={post.id}>
8           <Link
9             href={{
10               pathname: '/blog/[slug]',
11               query: { slug: post.slug },
12             }}
13           >
14             {post.title}
15           </Link>
16         </li>
17       )));
18     </ul>
19   );
20 }
```

```
21  
22 export default Posts;
```

Now, instead of using interpolation to create the path, we use a URL object in `href` where:

- `pathname` is the name of the page in the `pages` directory. `/blog/[slug]` in this case.
- `query` is an object with the dynamic segment. `slug` in this case.

Injecting the router

► Examples

To access the `router` object in a React component you can use `useRouter` or `withRouter`.

In general we recommend using `useRouter`.

Imperative Routing

► Examples

`next/link` should be able to cover most of your routing needs, but you can also do client-side navigations without it, take a look at the documentation for `next/router`.

The following example shows how to do basic page navigations with `useRouter`:

```
1 import { useRouter } from 'next/router';
2
3 export default function ReadMore() {
4   const router = useRouter();
5
6   return (
7     <button onClick={() => router.push('/about')}>
8       Click here to read more
9     </button>
10    );
11 }
```

Shallow Routing

► Examples

Shallow routing allows you to change the URL without running data fetching methods again, that includes `getServerSideProps`, `getStaticProps`, and `getInitialProps`.

You'll receive the updated `pathname` and the `query` via the `router` object (added by `useRouter` or `withRouter`), without losing state.

To enable shallow routing, set the `shallow` option to `true`. Consider the following example:

```
1 import { useEffect } from 'react';
2 import { useRouter } from 'next/router';
3
4 // Current URL is '/'
5 function Page() {
6   const router = useRouter();
7
8   useEffect(() => {
9     // Always do navigations after the first render
10    router.push('/?counter=10', undefined, { shallow: true });
11  }, []);
12
13  useEffect(() => {
14    // The counter changed!
15  }, [router.query.counter]);
16}
17
18 export default Page;
```

The URL will get updated to `/?counter=10`. and the page won't get replaced, only the state of the route is changed.

You can also watch for URL changes via `componentDidUpdate` ↗ as shown below:

```
1 componentDidUpdate(prevProps) {
2   const { pathname, query } = this.props.router
3   // verify props have changed to avoid an infinite loop
4   if (query.counter !== prevProps.router.query.counter) {
5     // fetch data based on the new query
6   }
7 }
```

Caveats

Shallow routing **only** works for URL changes in the current page. For example, let's assume we have another page called `pages/about.js`, and you run this:

```
router.push('/?counter=10', '/about?counter=10', { shallow: true });
```

Since that's a new page, it'll unload the current page, load the new one and wait for data fetching even though we asked to do shallow routing.

When shallow routing is used with middleware it will not ensure the new page matches the current page like previously done without middleware. This is due to middleware being able to rewrite dynamically and can't be verified client-side without a data fetch which is skipped with shallow, so a shallow route change must always be treated as shallow.

> Menu

Pages Router > ... > Routing > Middleware

Middleware

Middleware allows you to run code before a request is completed. Then, based on the incoming request, you can modify the response by rewriting, redirecting, modifying the request or response headers, or responding directly.

Middleware runs before cached content and routes are matched. See [Matching Paths](#) for more details.

Convention

Use the file `middleware.ts` (or `.js`) in the root of your project to define Middleware. For example, at the same level as `pages` or `app`, or inside `src` if applicable.

Example

ts middleware.ts



```
1 import { NextResponse } from 'next/server';
2 import type { NextRequest } from 'next/server';
3
4 // This function can be marked `async` if using `await` inside
5 export function middleware(request: NextRequest) {
6   return NextResponse.redirect(new URL('/home', request.url));
7 }
8
9 // See "Matching Paths" below to learn more
10 export const config = {
11   matcher: '/about/:path*',
12 };
```

Matching Paths

Middleware will be invoked for **every route in your project**. The following is the execution order:

1. `headers` from `next.config.js`
2. `redirects` from `next.config.js`
3. Middleware (`rewrites`, `redirects`, etc.)
4. `beforeFiles` (`rewrites`) from `next.config.js`
5. Filesystem routes (`public/`, `_next/static/`, `pages/`, `app/`, etc.)
6. `afterFiles` (`rewrites`) from `next.config.js`
7. Dynamic Routes (`/blog/[slug]`)
8. `fallback` (`rewrites`) from `next.config.js`

There are two ways to define which paths Middleware will run on:

1. [Custom matcher config](#)
2. [Conditional statements](#)

Matcher

`matcher` allows you to filter Middleware to run on specific paths.

`middleware.js`

```
1 export const config = {
2   matcher: '/about/:path*',
3 };
```

You can match a single path or multiple paths with an array syntax:

`middleware.js`

```
1 export const config = {
2   matcher: ['/about/:path*', '/dashboard/:path*'],
3 };
```

The `matcher` config allows full regex so matching like negative lookaheads or character matching is supported. An example of a negative lookahead to match all except specific paths can be seen here:



```

1  export const config = {
2    matcher: [
3      /*
4        * Match all request paths except for the ones starting with:
5        * - api (API routes)
6        * - _next/static (static files)
7        * - _next/image (image optimization files)
8        * - favicon.ico (favicon file)
9        */
10       '/((?!api|_next/static|_next/image|favicon.ico).*)',
11     ],
12   };

```

Note: The `matcher` values need to be constants so they can be statically analyzed at build-time. Dynamic values such as variables will be ignored.

Configured matchers:

1. MUST start with `/`
2. Can include named parameters: `/about/:path` matches `/about/a` and `/about/b` but not `/about/a/c`
3. Can have modifiers on named parameters (starting with `:`): `/about/:path*` matches `/about/a/b/c` because `*` is zero or more. `?` is zero or one and `+` one or more
4. Can use regular expression enclosed in parenthesis: `/about/(.*)` is the same as `/about/:path*`

Read more details on [path-to-regexp](#) documentation.

Note: For backward compatibility, Next.js always considers `/public` as `/public/index`. Therefore, a matcher of `/public/:path` will match.

Conditional Statements



```

1  import { NextResponse } from 'next/server';
2  import type { NextRequest } from 'next/server';
3
4  export function middleware(request: NextRequest) {
5    if (request.nextUrl.pathname.startsWith('/about')) {
6      return NextResponse.rewrite(new URL('/about-2', request.url));
7    }

```

```
8
9  if (request.nextUrl.pathname.startsWith('/dashboard')) {
10    return NextResponse.rewrite(new URL('/dashboard/user', request.url));
11  }
12 }
```

NextResponse

The `NextResponse` API allows you to:

- `redirect` the incoming request to a different URL
- `rewrite` the response by displaying a given URL
- Set request headers for API Routes, `getServerSideProps`, and `rewrite` destinations
- Set response cookies
- Set response headers

To produce a response from Middleware, you can:

1. `rewrite` to a route ([Page](#) or [Edge API Route](#)) that produces a response
2. return a `NextResponse` directly. See [Producing a Response](#)

Using Cookies

Cookies are regular headers. On a `Request`, they are stored in the `Cookie` header. On a `Response` they are in the `Set-Cookie` header. Next.js provides a convenient way to access and manipulate these cookies through the `cookies` extension on `NextRequest` and `NextResponse`.

1. For incoming requests, `cookies` comes with the following methods: `get`, `getAll`, `set`, and `delete`. You can check for the existence of a cookie with `has` or remove all cookies with `clear`.
2. For outgoing responses, `cookies` have the following methods `get`, `getAll`, `set`, and `delete`.

TS middleware.ts



```
1 import { NextResponse } from 'next/server';
2 import type { NextRequest } from 'next/server';
3
4 export function middleware(request: NextRequest) {
```

```
5 // Assume a "Cookie:nextjs=fast" header to be present on the incoming request
6 // Getting cookies from the request using the `RequestCookies` API
7 let cookie = request.cookies.get('nextjs')?.value;
8 console.log(cookie); // => 'fast'
9 const allCookies = request.cookies.getAll();
10 console.log(allCookies); // => [{ name: 'nextjs', value: 'fast' }]
11
12 request.cookies.has('nextjs'); // => true
13 request.cookies.delete('nextjs');
14 request.cookies.has('nextjs'); // => false
15
16 // Setting cookies on the response using the `ResponseCookies` API
17 const response = NextResponse.next();
18 response.cookies.set('vercel', 'fast');
19 response.cookies.set({
20   name: 'vercel',
21   value: 'fast',
22   path: '/',
23 });
24 cookie = response.cookies.get('vercel');
25 console.log(cookie); // => { name: 'vercel', value: 'fast', Path: '/' }
26 // The outgoing response will have a `Set-Cookie:vercel=fast;path=/test` header.
27
28 return response;
29 }
```

Setting Headers

You can set request and response headers using the `NextResponse` API (setting `request` headers is available since Next.js v13.0.0).

```
TS middleware.ts
```

```
1 import { NextResponse } from 'next/server';
2 import type { NextRequest } from 'next/server';
3
4 export function middleware(request: NextRequest) {
5   // Clone the request headers and set a new header `x-hello-from-middleware1`
6   const requestHeaders = new Headers(request.headers);
7   requestHeaders.set('x-hello-from-middleware1', 'hello');
8
9   // You can also set request headers in NextResponse.rewrite
10  const response = NextResponse.next({
11    request: {
12      // New request headers
13      headers: requestHeaders,
14    },
15  });
}
```

```
16
17 // Set a new response header `x-hello-from-middleware2`
18 response.headers.set('x-hello-from-middleware2', 'hello');
19 return response;
20 }
```

Note: Avoid setting large headers as it might cause [431 Request Header Fields Too Large ↗](#) error depending on your backend web server configuration.

Producing a Response

You can respond from Middleware directly by returning a `Response` or `NextResponse` instance. (This is available since [Next.js v13.1.0 ↗](#))

middleware.ts

```
1 import { NextRequest, NextResponse } from 'next/server';
2 import { isAuthenticated } from '@lib/auth';
3
4 // Limit the middleware to paths starting with `/api/`
5 export const config = {
6   matcher: '/api/:function*',
7 };
8
9 export function middleware(request: NextRequest) {
10   // Call our authentication function to check the request
11   if (!isAuthenticated(request)) {
12     // Respond with JSON indicating an error message
13     return new NextResponse(
14       JSON.stringify({ success: false, message: 'authentication failed' }),
15       { status: 401, headers: { 'content-type': 'application/json' } },
16     );
17   }
18 }
```

Advanced Middleware Flags

In [v13.1](#) of Next.js two additional flags were introduced for middleware, `skipMiddlewareUrlNormalize` and `skipTrailingSlashRedirect` to handle advanced use cases.

`skipTrailingSlashRedirect` allows disabling Next.js default redirects for adding or removing trailing slashes allowing custom handling inside middleware which can allow maintaining the trailing slash for some paths but not others allowing easier incremental migrations.

JS next.config.js

```
1 module.exports = {
2   skipTrailingSlashRedirect: true,
3 };
```

JS middleware.js

```
1 const legacyPrefixes = ['/docs', '/blog'];
2
3 export default async function middleware(req) {
4   const { pathname } = req.nextUrl;
5
6   if (legacyPrefixes.some((prefix) => pathname.startsWith(prefix))) {
7     return NextResponse.next();
8   }
9
10  // apply trailing slash handling
11  if (
12    !pathname.endsWith('/') &&
13    !pathname.match(/((?!\.well-known(?:\.*))(:[^\/]+\/)*[^\/]+\.\w+)/)
14  ) {
15    req.nextUrl.pathname += '/';
16    return NextResponse.redirect(req.nextUrl);
17  }
18 }
```

`skipMiddlewareUrlNormalize` allows disabling the URL normalizing Next.js does to make handling direct visits and client-transitions the same. There are some advanced cases where you need full control using the original URL which this unlocks.

JS next.config.js

```
1 module.exports = {
2   skipMiddlewareUrlNormalize: true,
3 };
```

JS middleware.js

```
1 export default async function middleware(req) {
2   const { pathname } = req.nextUrl;
3 }
```

```
4 // GET /_next/data/build-id/hello.json
5
6 console.log(pathname);
7 // with the flag this now /_next/data/build-id/hello.json
8 // without the flag this would be normalized to /hello
9 }
```

Version History

Version	Changes
v13.1.0	Advanced Middleware flags added
v13.0.0	Middleware can modify request headers, response headers, and send responses
v12.2.0	Middleware is stable, please see the upgrade guide
v12.0.9	Enforce absolute URLs in Edge Runtime (PR ↗)
v12.0.0	Middleware (Beta) added

> Menu

Pages Router > ... > Routing > Pages and Layouts

Pages and Layouts

The Pages Router has a file-system based router built on the [concept of pages](#).

When a file is added to the `pages` directory, it's automatically available as a route.

In Next.js, a **page** is a [React Component](#) ↗ exported from a `.js`, `.jsx`, `.ts`, or `.tsx` file in the `pages` directory. Each page is associated with a route based on its file name.

Example: If you create `pages/about.js` that exports a React component like below, it will be accessible at `/about`.

```
1 export default function About() {  
2   return <div>About</div>;  
3 }
```

Index routes

The router will automatically route files named `index` to the root of the directory.

- `pages/index.js` → `/`
- `pages/blog/index.js` → `/blog`

Nested routes

The router supports nested files. If you create a nested folder structure, files will automatically be routed in the same way still.

- `pages/blog/first-post.js` → `/blog/first-post`
- `pages/dashboard/settings/username.js` → `/dashboard/settings/username`

Pages with Dynamic Routes

Next.js supports pages with dynamic routes. For example, if you create a file called `pages/posts/[id].js`, then it will be accessible at `posts/1`, `posts/2`, etc.

To learn more about dynamic routing, check the [Dynamic Routing documentation](#).

Layout Pattern

The React model allows us to deconstruct a [page](#) into a series of components. Many of these components are often reused between pages. For example, you might have the same navigation bar and footer on every page.

`JS` `components/layout.js`

```
1 import Navbar from './navbar';
2 import Footer from './footer';
3
4 export default function Layout({ children }) {
5   return (
6     <>
7       <Navbar />
8       <main>{children}</main>
9       <Footer />
10    </>
11  );
12}
```

Examples

Single Shared Layout with Custom App

If you only have one layout for your entire application, you can create a [Custom App](#) and wrap your application with the layout. Since the `<Layout />` component is re-used when changing pages, its component state will be preserved (e.g. input values).

JS pages/_app.js

```
1 import Layout from '../components/layout';
2
3 export default function MyApp({ Component, pageProps }) {
4   return (
5     <Layout>
6       <Component {...pageProps} />
7     </Layout>
8   );
9 }
```

Per-Page Layouts

If you need multiple layouts, you can add a property `getLayout` to your page, allowing you to return a React component for the layout. This allows you to define the layout on a *per-page basis*. Since we're returning a function, we can have complex nested layouts if desired.

JS pages/index.js

```
1
2 import Layout from '../components/layout'
3 import NestedLayout from '../components/nested-layout'
4
5 export default function Page() {
6   return (
7     /** Your content */
8   )
9 }
10
11 Page.getLayout = function getLayout(page) {
12   return (
13     <Layout>
14       <NestedLayout>{page}</NestedLayout>
15     </Layout>
16   )
17 }
```

JS pages/_app.js

```
1 export default function MyApp({ Component, pageProps }) {
2   // Use the layout defined at the page level, if available
3   const getLayout = Component.getLayout || ((page) => page);
```

```
4
5     return getLayout(<Component {...pageProps} />);
6 }
```

When navigating between pages, we want to *persist* page state (input values, scroll position, etc.) for a Single-Page Application (SPA) experience.

This layout pattern enables state persistence because the React component tree is maintained between page transitions. With the component tree, React can understand which elements have changed to preserve state.

Note: This process is called [reconciliation](#), which is how React understands which elements have changed.

With TypeScript

When using TypeScript, you must first create a new type for your pages which includes a `getLayout` function. Then, you must create a new type for your `AppProps` which overrides the `Component` property to use the previously created type.

pages/index.tsx

```
1 import type { ReactElement } from 'react';
2 import Layout from '../components/layout';
3 import NestedLayout from '../components/nested-layout';
4 import type { NextPageWithLayout } from './_app';
5
6 const Page: NextPageWithLayout = () => {
7     return <p>hello world</p>;
8 };
9
10 Page.getLayout = function getLayout(page: ReactElement) {
11     return (
12         <Layout>
13             <NestedLayout>{page}</NestedLayout>
14         </Layout>
15     );
16 };
17
18 export default Page;
```

pages/_app.tsx

```
1 import type { ReactElement, ReactDOMNode } from 'react';
2 import type { NextPage } from 'next';
3 import type { AppProps } from 'next/app';
4
```

```
5 export type NextPageWithLayout<P = {}, IP = P> = NextPage<P, IP> & {
6   getLayout?: (page: ReactElement) => ReactNode;
7 }
8
9 type AppPropsWithLayout = AppProps & {
10   Component: NextPageWithLayout;
11 }
12
13 export default function MyApp({ Component, pageProps }: AppPropsWithLayout) {
14   // Use the layout defined at the page level, if available
15   const getLayout = Component.getLayout ?? ((page) => page);
16
17   return getLayout(<Component {...pageProps} />);
18 }
```

Data Fetching

Inside your layout, you can fetch data on the client-side using `useEffect` or a library like [SWR](#). Because this file is not a `Page`, you cannot use `getStaticProps` or `getServerSideProps` currently.

JS components/layout.js

```
1 import useSWR from 'swr';
2 import Navbar from './navbar';
3 import Footer from './footer';
4
5 export default function Layout({ children }) {
6   const { data, error } = useSWR('/api/navigation', fetcher);
7
8   if (error) return <div>Failed to load</div>;
9   if (!data) return <div>Loading...</div>;
10
11   return (
12     <>
13       <Navbar links={data.links} />
14       <main>{children}</main>
15       <Footer />
16     </>
17   );
18 }
```

> Menu

Pages Router > Building Your Application > Styling

Styling

Next.js supports different ways of styling your application, including:

- **Global CSS:** Simple to use and familiar for those experienced with traditional CSS, but can lead to larger CSS bundles and difficulty managing styles as the application grows.
- **CSS Modules:** Create locally scoped CSS classes to avoid naming conflicts and improve maintainability.
- **Tailwind CSS:** A utility-first CSS framework that allows for rapid custom designs by composing utility classes.
- **Sass:** A popular CSS preprocessor that extends CSS with features like variables, nested rules, and mixins.
- **CSS-in-JS:** Embed CSS directly in your JavaScript components, enabling dynamic and scoped styling.

Learn more about each approach by exploring their respective documentation:

[CSS Modules](#)

Style your Next.js Application using CSS Modules.

[Tailwind CSS](#)

Style your Next.js Application using Tailwind CSS.

[CSS-in-JS](#)

Use CSS-in-JS libraries with Next.js

[Sass](#)

Learn how to use Sass in your Next.js application.

> Menu

Pages Router > ... > Styling > CSS-in-JS

CSS-in-JS

► Examples

It's possible to use any existing CSS-in-JS solution. The simplest one is inline styles:

```
1 function HiThere() {
2   return <p style={{ color: 'red' }}>hi there</p>;
3 }
4
5 export default HiThere;
```

We bundle [styled-jsx](#) to provide support for isolated scoped CSS. The aim is to support "shadow CSS" similar to Web Components, which unfortunately [do not support server-rendering and are JS-only](#).

See the above examples for other popular CSS-in-JS solutions (like Styled Components).

A component using `styled-jsx` looks like this:

```
1 function HelloWorld() {
2   return (
3     <div>
4       Hello world
5       <p>scoped!</p>
6       <style jsx>{
7         p {
8           color: blue;
9         }
10        div {
11          background: red;
12        }
13        @media (max-width: 600px) {
14          div {
15            background: blue;
16          }
17        }
18      `}</style>
19      <style global jsx>{
20        body {
```

```
21         background: black;
22     }
23 `}</style>
24 </div>
25 );
26 }
27
28 export default HelloWorld;
```

Please see the [styled-jsx documentation](#) ↗ for more examples.

Disabling JavaScript

Yes, if you disable JavaScript the CSS will still be loaded in the production build (`next start`). During development, we require JavaScript to be enabled to provide the best developer experience with [Fast Refresh](#) ↗.

> Menu

Pages Router > ... > Styling > CSS Modules

CSS Modules

▼ Examples

- [Basic CSS Example ↗](#)

Next.js has built-in support for CSS Modules using the `.module.css` extension.

CSS Modules locally scope CSS by automatically creating a unique class name. This allows you to use the same class name in different files without worrying about collisions. This behavior makes CSS Modules the ideal way to include component-level CSS.

Example

For example, consider a reusable `Button` component in the `components/` folder:

First, create `components/Button.module.css` with the following content:

📄 Button.module.css



```
1  /*
2   You do not need to worry about .error {} colliding with any other `*.css` or
3   `*.module.css` files!
4  */
5  .error {
6    color: white;
7    background-color: red;
8  }
```

Then, create `components/Button.js`, importing and using the above CSS file:

JS components/Button.js



```
1 import styles from './Button.module.css';
2
3 export function Button() {
4   return (
5     <button
6       type="button"
7       // Note how the "error" class is accessed as a property on the imported
8       // `styles` object.
9       className={styles.error}
10    >
11      Destroy
12    </button>
13  );
14}
```

CSS Modules are an *optional feature* and are **only enabled for files with the `.module.css` extension**. Regular `<link>` stylesheets and global CSS files are still supported.

In production, all CSS Module files will be automatically concatenated into **many minified and code-split `.css` files**. These `.css` files represent hot execution paths in your application, ensuring the minimal amount of CSS is loaded for your application to paint.

Global Styles

To add a stylesheet to your application, import the CSS file within `pages/_app.js`.

For example, consider the following stylesheet named `styles.css`:

styles.css

```
1 body {
2   font-family: 'SF Pro Text', 'SF Pro Icons', 'Helvetica Neue', 'Helvetica',
3   'Arial', sans-serif;
4   padding: 20px 20px 60px;
5   max-width: 680px;
6   margin: 0 auto;
7 }
```

Create a `pages/_app.js` file if not already present. Then, `import` ↗ the `styles.css` file.

pages/_app.js

```
1 import '../styles.css';
```

```
2
3 // This default export is required in a new `pages/_app.js` file.
4 export default function MyApp({ Component, pageProps }) {
5   return <Component {...pageProps} />;
6 }
```

These styles (`styles.css`) will apply to all pages and components in your application. Due to the global nature of stylesheets, and to avoid conflicts, you may **only import them inside `pages/_app.js`**.

In development, expressing stylesheets this way allows your styles to be hot reloaded as you edit them—meaning you can keep application state.

In production, all CSS files will be automatically concatenated into a single minified `.css` file.

External Stylesheets

Next.js allows you to import CSS files from a JavaScript file. This is possible because Next.js extends the concept of [import ↗](#) beyond JavaScript.

Import styles from `node_modules`

Since Next.js **9.5.4**, importing a CSS file from `node_modules` is permitted anywhere in your application.

For global stylesheets, like `bootstrap` or `nprogress`, you should import the file inside `pages/_app.js`. For example:

JS `pages/_app.js`

```
1 import 'bootstrap/dist/css/bootstrap.css';
2
3 export default function MyApp({ Component, pageProps }) {
4   return <Component {...pageProps} />;
5 }
```

For importing CSS required by a third-party component, you can do so in your component. For example:

JS `components/example-dialog.js`

```
1 import { useState } from 'react';
2 import { Dialog } from '@reach/dialog';
3 import VisuallyHidden from '@reach/visually-hidden';
4 import '@reach/dialog/styles.css';
```

```
5
6 function ExampleDialog(props) {
7   const [showDialog, setShowDialog] = useState(false);
8   const open = () => setShowDialog(true);
9   const close = () => setShowDialog(false);
10
11  return (
12    <div>
13      <button onClick={open}>Open Dialog</button>
14      <Dialog isOpen={showDialog} onDismiss={close}>
15          <button className="close-button" onClick={close}>
16              <VisuallyHidden>Close</VisuallyHidden>
17              <span aria-hidden>x</span>
18          </button>
19          <p>Hello there. I am a dialog</p>
20      </Dialog>
21    </div>
22  );
23 }
```

Additional Features

Next.js includes additional features to improve the authoring experience of adding styles:

- When running locally with `next dev`, local stylesheets (either global or CSS modules) will take advantage of **Fast Refresh** to instantly reflect changes as edits are saved.
- When building for production with `next build`, CSS files will be bundled into fewer minified `.css` files to reduce the number of network requests needed to retrieve styles.
- If you disable JavaScript, styles will still be loaded in the production build (`next start`). However, JavaScript is still required for `next dev` to enable **Fast Refresh**.

> Menu

Pages Router > ... > Styling > Sass

Sass

Next.js has built-in support for Sass using both the `.scss` and `.sass` extensions. You can use component-level Sass via CSS Modules and the `.module.scss` or `.module.sass` extension.

First, install `sass` ↗:

> Terminal

```
npm install --save-dev sass
```

Good to know:

Sass supports [two different syntax](#) ↗, each with their own extension. The `.scss` extension requires you use the [SCSS syntax](#) ↗, while the `.sass` extension requires you use the [Indented Syntax \("Sass"\)](#) ↗.

If you're not sure which to choose, start with the `.scss` extension which is a superset of CSS, and doesn't require you learn the Indented Syntax ("Sass").

Customizing Sass Options

If you want to configure the Sass compiler, use `sassOptions` in `next.config.js`.

js next.config.js

```
1 const path = require('path');
2
3 module.exports = {
4   sassOptions: {
5     includePaths: [path.join(__dirname, 'styles')],
6   },
7 }
```

Sass Variables

Next.js supports Sass variables exported from CSS Module files.

For example, using the exported `primaryColor` Sass variable:

📄 app/variables.module.scss

```
1 $primary-color: #64ff00;
2
3 :export {
4   primaryColor: $primary-color;
5 }
```

📄 pages/_app.js

```
1 import variables from '../styles/variables.module.scss';
2
3 export default function MyApp({ Component, pageProps }) {
4   return (
5     <Layout color={variables.primaryColor}>
6       <Component {...pageProps} />
7     </Layout>
8   );
9 }
```

> Menu

Pages Router > ... > Styling > Tailwind CSS

Tailwind CSS

▼ Examples

- [With Tailwind CSS ↗](#)

[Tailwind CSS ↗](#) is a utility-first CSS framework that works exceptionally well with Next.js.

Installing Tailwind

Install the Tailwind CSS packages and run the `init` command to generate both the `tailwind.config.js` and `postcss.config.js` files:

> Terminal



```
1 npm install -D tailwindcss postcss autoprefixer
2 npx tailwindcss init -p
```

Configuring Tailwind

Inside `tailwind.config.js`, add paths to the files that will use Tailwind CSS class names:

& tailwind.config.js



```
1 /** @type {import('tailwindcss').Config} */
2 module.exports = {
3   content: [
4     './app/**/*.{js,ts,jsx,tsx,mdx}', // Note the addition of the `app` directory.
5     './pages/**/*.{js,ts,jsx,tsx,mdx}',
```

```
6   './components/**/*.{js,ts,jsx,tsx,mdx}',  
7  
8     // Or if using `src` directory:  
9     './src/**/*.{js,ts,jsx,tsx,mdx}',  
10    ],  
11    theme: {  
12      extend: {},  
13    },  
14    plugins: [],  
15  };
```

You do not need to modify `postcss.config.js`.

Importing Styles

Add the [Tailwind CSS directives](#) that Tailwind will use to inject its generated styles to a [Global Stylesheet](#) in your application, for example:

 app/globals.css 

```
1 @tailwind base;  
2 @tailwind components;  
3 @tailwind utilities;
```

Inside the [root layout](#) (`app/layout.tsx`), import the `globals.css` stylesheet to apply the styles to every route in your application.

 app/layout.tsx 

```
1 import type { Metadata } from 'next';  
2  
3 // These styles apply to every route in the application  
4 import './globals.css';  
5  
6 export const metadata: Metadata = {  
7   title: 'Create Next App',  
8   description: 'Generated by create next app',  
9 };  
10  
11 export default function RootLayout({  
12   children,  
13 }: {  
14   children: React.ReactNode;  
15 }) {  
16   return (
```

```
17     <html lang="en">
18         <body>{children}</body>
19     </html>
20 );
21 }
```

Using Classes

After installing Tailwind CSS and adding the global styles, you can use Tailwind's utility classes in your application.

app/page.tsx

```
1 export default function Page() {
2     return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>;
3 }
```

Usage with Turbopack

As of Next.js 13.1, Tailwind CSS and PostCSS are supported with [Turbopack ↗](#).

> Menu

Pages Router > Building Your Application > Upgrading

Upgrading

Upgrade your application to newer versions of Next.js or migrate from the Pages Router to the App Router.

Codemods

Use codemods to upgrade your Next.js codebase when new features are released.

From Pages to App

Learn how to upgrade your existing Next.js application from the Pages Router to the App Router.

Version 13

Upgrade your Next.js Application from Version 12 to 13.

Version 12

Upgrade your Next.js Application from Version 11 to Version 12.

Version 11

Upgrade your Next.js Application from Version 10 to Version 11.

Version 10

Upgrade your Next.js Application from Version 9 to Version 10.

Version 9

Upgrade your Next.js Application from Version 8 to Version 9.

> Menu

Pages Router > ... > Upgrading > From Pages to App

From Pages to App

This guide will help you:

- [Update your Next.js application from version 12 to version 13](#)
- [Upgrade features that work in both the `pages` and the `app` directories](#)
- [Incrementally migrate your existing application from `pages` to `app`](#)

Upgrading

Node.js Version

The minimum Node.js version is now **v16.8**. See the [Node.js documentation](#) for more information.

Next.js Version

To update to Next.js version 13, run the following command using your preferred package manager:

>_ Terminal



```
npm install next@latest react@latest react-dom@latest
```

ESLint Version

If you're using ESLint, you need to upgrade your ESLint version:

>_ Terminal



```
npm install -D eslint-config-next@latest
```

Note: You may need to restart the ESLint server in VS Code for the ESLint changes to take effect. Open the Command Palette (`cmd+shift+p` on Mac; `ctrl+shift+p` on Windows) and search for `ESLint: Restart ESLint Server`.

Next Steps

After you've updated, see the following sections for next steps:

- [Upgrade new features](#): A guide to help you upgrade to new features such as the improved Image and Link Components.
- [Migrate from the `pages` to `app` directory](#): A step-by-step guide to help you incrementally migrate from the `pages` to the `app` directory.

Upgrading New Features

Next.js 13 introduced the new [App Router](#) with new features and conventions. The new Router is available in the `app` directory and co-exists with the `pages` directory.

Upgrading to Next.js 13 does **not** require using the new [App Router](#). You can continue using `pages` with new features that work in both directories, such as the updated [Image component](#), [Link component](#), [Script component](#), and [Font optimization](#).

`<Image/>` Component

Next.js 12 introduced new improvements to the Image Component with a temporary import: `next/future/image`. These improvements included less client-side JavaScript, easier ways to extend and style images, better accessibility, and native browser lazy loading.

In version 13, this new behavior is now the default for `next/image`.

There are two codemods to help you migrate to the new Image Component:

- [`next-image-to-legacy-image` codemod](#): Safely and automatically renames `next/image` imports to `next/legacy/image`. Existing components will maintain the same behavior.
- [`next-image-experimental` codemod](#): Dangerously adds inline styles and removes unused props. This will change the behavior of existing components to match the new defaults. To use this codemod, you need to run the `next-image-to-legacy-image` codemod first.

<Link> Component

The `<Link>` Component no longer requires manually adding an `<a>` tag as a child. This behavior was added as an experimental option in [version 12.2](#) and is now the default. In Next.js 13, `<Link>` always renders `<a>` and allows you to forward props to the underlying tag.

For example:

```
1 import Link from 'next/link'  
2  
3 // Next.js 12: `<a>` has to be nested otherwise it's excluded  
4 <Link href="/about">  
5   <a>About</a>  
6 </Link>  
7  
8 // Next.js 13: `<Link>` always renders `<a>` under the hood  
9 <Link href="/about">  
10   About  
11 </Link>
```

To upgrade your links to Next.js 13, you can use the [new-link codemod](#).

<Script> Component

The behavior of `next/script` has been updated to support both `pages` and `app`, but some changes need to be made to ensure a smooth migration:

- Move any `beforeInteractive` scripts you previously included in `_document.js` to the root layout file (`app/layout.tsx`).
- The experimental `worker` strategy does not yet work in `app` and scripts denoted with this strategy will either have to be removed or modified to use a different strategy (e.g. `lazyOnload`).
- `onLoad`, `onReady`, and `onError` handlers will not work in Server Components so make sure to move them to a [Client Component](#) or remove them altogether.

Font Optimization

Previously, Next.js helped you optimize fonts by [inlining font CSS](#). Version 13 introduces the new `next/font` module which gives you the ability to customize your font loading experience while still ensuring great performance and privacy. `next/font` is supported in both the `pages` and `app` directories.

While [inlining CSS](#) still works in `pages`, it does not work in `app`. You should use `next/font` instead.

See the [Font Optimization](#) page to learn how to use `next/font`.

Migrating from `pages` to `app`

Moving to the App Router may be the first time using React features that Next.js builds on top of such as Server Components, Suspense, and more. When combined with new Next.js features such as [special files](#) and [layouts](#), migration means new concepts, mental models, and behavioral changes to learn.

We recommend reducing the combined complexity of these updates by breaking down your migration into smaller steps. The `app` directory is intentionally designed to work simultaneously with the `pages` directory to allow for incremental page-by-page migration.

- The `app` directory supports nested routes *and* layouts. [Learn more](#).
- Use nested folders to [define routes](#) and a special `page.js` file to make a route segment publicly accessible. [Learn more](#).
- [Special file conventions](#) are used to create UI for each route segment. The most common special files are `page.js` and `layout.js`.
 - Use `page.js` to define UI unique to a route.
 - Use `layout.js` to define UI that is shared across multiple routes.
 - `.js`, `.jsx`, or `.tsx` file extensions can be used for special files.
- You can colocate other files inside the `app` directory such as components, styles, tests, and more. [Learn more](#).
- Data fetching functions like `getServerSideProps` and `getStaticProps` have been replaced with [a new API](#) inside `app`. `getStaticPaths` has been replaced with `generateStaticParams`.
- `pages/_app.js` and `pages/_document.js` have been replaced with a single `app/layout.js` root layout. [Learn more](#).
- `pages/_error.js` has been replaced with more granular `error.js` special files. [Learn more](#).
- `pages/404.js` has been replaced with the `not-found.js` file.
- You can colocate other files inside the `app` directory such as components, styles, tests, and more. [Learn more](#).
- `pages/api/*` currently remain inside the `pages` directory.

Step 1: Creating the `app` directory

Update to the latest Next.js version (requires 13.4 or greater):

```
npm install next@latest
```

Then, create a new `app` directory at the root of your project (or `src/` directory).

Step 2: Creating a Root Layout

Create a new `app/layout.tsx` file inside the `app` directory. This is a [root layout](#) that will apply to all routes inside `app`.

TS app/layout.tsx

```
1 export default function RootLayout({  
2   // Layouts must accept a children prop.  
3   // This will be populated with nested layouts or pages  
4   children,  
5 }: {  
6   children: React.ReactNode;  
7 }) {  
8   return (  
9     <html lang="en">  
10    <body>{children}</body>  
11    </html>  
12  );  
13}
```

- The `app` directory **must** include a root layout.
- The root layout must define `<html>`, and `<body>` tags since Next.js does not automatically create them
- The root layout replaces the `pages/_app.tsx` and `pages/_document.tsx` files.
- `.js`, `.jsx`, or `.tsx` extensions can be used for layout files.

To manage `<head>` HTML elements, you can use the [built-in SEO support](#):

TS app/layout.tsx

```
1 import { Metadata } from 'next';  
2  
3 export const metadata: Metadata = {  
4   title: 'Home',  
5   description: 'Welcome to Next.js',  
6};
```

Migrating `_document.js` and `_app.js`

If you have an existing `_app` or `_document` file, you can copy the contents (e.g. global styles) to the root layout (`app/layout.tsx`). Styles in `app/layout.tsx` will *not* apply to `pages/*`. You should keep `_app` / `_document` while migrating to prevent your `pages/*` routes from breaking. Once fully migrated, you can then safely delete them.

If you are using any React Context providers, they will need to be moved to a [Client Component](#).

Migrating the `getLayout()` pattern to Layouts (Optional)

Next.js recommended adding a [property to Page components](#) to achieve per-page layouts in the `pages` directory. This pattern can be replaced with native support for [nested layouts](#) in the `app` directory.

► See before and after example

Step 3: Migrating `next/head`

In the `pages` directory, the `next/head` React component is used to manage `<head>` HTML elements such as `title` and `meta`. In the `app` directory, `next/head` is replaced with the new [built-in SEO support](#).

Before:

```
TS pages/index.tsx

1 import Head from 'next/head';
2
3 export default function Page() {
4   return (
5     <>
6       <Head>
7         <title>My page title</title>
8       </Head>
9     </>
10   );
11 }
```

After:

```
TS app/page.tsx

1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: 'My Page Title',
5 };
6
7 export default function Page() {
8   return '...';
9 }
```

[See all metadata options](#).

Step 4: Migrating Pages

- Pages in the `app` directory are **Server Components** by default. This is different from the `pages` directory where pages are **Client Components**.
- **Data fetching** has changed in `app`. `getServerSideProps`, `getStaticProps` and `getInitialProps` have been replaced for a simpler API.
- The `app` directory uses nested folders to **define routes** and a special `page.js` file to make a route segment publicly accessible.

pages Directory	app Directory	Route
<code>index.js</code>	<code>page.js</code>	<code>/</code>
<code>about.js</code>	<code>about/page.js</code>	<code>/about</code>
<code>blog/[slug].js</code>	<code>blog/[slug]/page.js</code>	<code>/blog/post-1</code>

We recommend breaking down the migration of a page into two main steps:

- Step 1: Move the default exported Page Component into a new Client Component.
- Step 2: Import the new Client Component into a new `page.js` file inside the `app` directory.

Note: This is the easiest migration path because it has the most comparable behavior to the `pages` directory.

Step 1: Create a new Client Component

- Create a new separate file inside the `app` directory (i.e. `app/home-page.tsx` or similar) that exports a Client Component. To define Client Components, add the `'use client'` directive to the top of the file (before any imports).
- Move the default exported page component from `pages/index.js` to `app/home-page.tsx`.

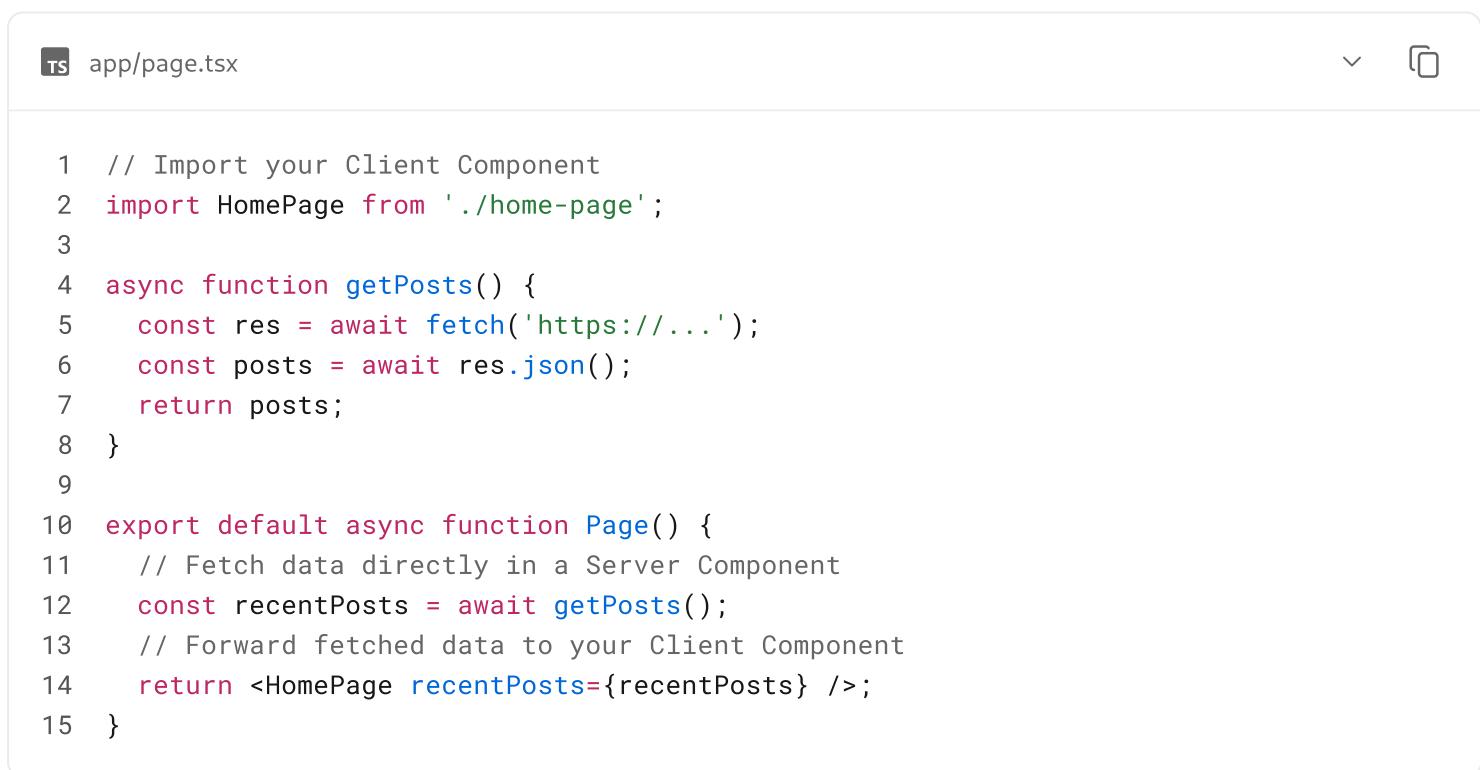
TS app/home-page.tsx

```
1  'use client';
2
3  // This is a Client Component. It receives data as props and
4  // has access to state and effects just like Page components
5  // in the `pages` directory.
6  export default function HomePage({ recentPosts }) {
7    return (
8      <div>
9        {recentPosts.map((post) => (
10          <div key={post.id}>{post.title}</div>
11        )))
12    )
13  }
```

```
12   </div>
13 );
14 }
```

Step 2: Create a new page

- Create a new `app/page.tsx` file inside the `app` directory. This is a Server Component by default.
- Import the `home-page.tsx` Client Component into the page.
- If you were fetching data in `pages/index.js`, move the data fetching logic directly into the Server Component using the new [data fetching APIs](#). See the [data fetching upgrade guide](#) for more details.



```
TS app/page.tsx ▾ ⌂

1 // Import your Client Component
2 import HomePage from './home-page';
3
4 async function getPosts() {
5   const res = await fetch('https://...');

6   const posts = await res.json();
7   return posts;
8 }
9
10 export default async function Page() {
11   // Fetch data directly in a Server Component
12   const recentPosts = await getPosts();
13   // Forward fetched data to your Client Component
14   return <HomePage recentPosts={recentPosts} />;
15 }
```

- If your previous page used `useRouter`, you'll need to update to the new routing hooks. [Learn more](#).
- Start your development server and visit <http://localhost:3000>. You should see your existing index route, now served through the app directory.

Step 5: Migrating Routing Hooks

A new router has been added to support the new behavior in the `app` directory.

In `app`, you should use the three new hooks imported from `next/navigation`: `useRouter()`, `usePathname()`, and `useSearchParams()`.

- The new `useRouter` hook is imported from `next/navigation` and has different behavior to the `useRouter` hook in `pages` which is imported from `next/router`.

- The `useRouter` hook imported from `next/router` is not supported in the `app` directory but can continue to be used in the `pages` directory.
- The new `useRouter` does not return the `pathname` string. Use the separate `usePathname` hook instead.
- The new `useRouter` does not return the `query` object. Use the separate `useSearchParams` hook instead.
- You can use `useSearchParams` and `usePathname` together to listen to page changes. See the [Router Events](#) section for more details.
- These new hooks are only supported in Client Components. They cannot be used in Server Components.

TS app/example-client-component.tsx

```

1  'use client';
2
3  import { useRouter, usePathname, useSearchParams } from 'next/navigation';
4
5  export default function ExampleClientComponent() {
6    const router = useRouter();
7    const pathname = usePathname();
8    const searchParams = useSearchParams();
9
10   // ...
11 }
```

In addition, the new `useRouter` hook has the following changes:

- `isFallback` has been removed because `fallback` has [been replaced](#).
- The `locale`, `locales`, `defaultLocales`, `domainLocales` values have been removed because built-in i18n Next.js features are no longer necessary in the `app` directory. [Learn more about i18n](#).
- `basePath` has been removed. The alternative will not be part of `useRouter`. It has not yet been implemented.
- `asPath` has been removed because the concept of `as` has been removed from the new router.
- `isReady` has been removed because it is no longer necessary. During [static rendering](#), any component that uses the `useSearchParams()` hook will skip the prerendering step and instead be rendered on the client at runtime.

[View the `useRouter\(\)` API reference](#).

Step 6: Migrating Data Fetching Methods

The `pages` directory uses `getServerSideProps` and `getStaticProps` to fetch data for pages. Inside the `app` directory, these previous data fetching functions are replaced with a [simpler API](#) built on top of `fetch()` and `async` React Server Components.

TS app/page.tsx

```
1 export default async function Page() {
2   // This request should be cached until manually invalidated.
3   // Similar to `getStaticProps`.
4   // `force-cache` is the default and can be omitted.
5   const staticData = await fetch(`https://...`, { cache: 'force-cache' });
6
7   // This request should be refetched on every request.
8   // Similar to `getServerSideProps`.
9   const dynamicData = await fetch(`https://...`, { cache: 'no-store' });
10
11  // This request should be cached with a lifetime of 10 seconds.
12  // Similar to `getStaticProps` with the `revalidate` option.
13  const revalidatedData = await fetch(`https://...`, {
14    next: { revalidate: 10 },
15  });
16
17  return <div>...</div>;
18 }
```

Server-side Rendering (`getServerSideProps`)

In the `pages` directory, `getServerSideProps` is used to fetch data on the server and forward props to the default exported React component in the file. The initial HTML for the page is prerendered from the server, followed by "hydrating" the page in the browser (making it interactive).

JS pages/dashboard.js

```
1 // `pages` directory
2
3 export async function getServerSideProps() {
4   const res = await fetch(`https://...`);
5   const projects = await res.json();
6
7   return { props: { projects } };
8 }
9
10 export default function Dashboard({ projects }) {
11   return (
12     <ul>
13       {projects.map((project) => (
14         <li key={project.id}>{project.name}</li>
15       )));
16     </ul>
17   );
}
```

In the `app` directory, we can colocate our data fetching inside our React components using [Server Components](#). This allows us to send less JavaScript to the client, while maintaining the rendered HTML from the server.

By setting the `cache` option to `no-store`, we can indicate that the fetched data should [never be cached](#). This is similar to `getServerSideProps` in the `pages` directory.

`TS` app/dashboard/page.tsx

```

1 // `app` directory
2
3 // This function can be named anything
4 async function getProjects() {
5   const res = await fetch(`https://...`, { cache: 'no-store' });
6   const projects = await res.json();
7
8   return projects;
9 }
10
11 export default async function Dashboard() {
12   const projects = await getProjects();
13
14   return (
15     <ul>
16       {projects.map((project) => (
17         <li key={project.id}>{project.name}</li>
18       ))}
19     </ul>
20   );
21 }
```

Accessing Request Object

In the `pages` directory, you can retrieve request-based data based on the Node.js HTTP API.

For example, you can retrieve the `req` object from `getServerSideProps` and use it to retrieve the request's cookies and headers.

`JS` pages/index.js

```

1 // `pages` directory
2
3 export async function getServerSideProps({ req, query }) {
4   const authHeader = req.getHeaders()['authorization'];
5   const theme = req.cookies['theme'];
6 }
```

```
7   return { props: { ... } }
8 }
9
10 export default function Page(props) {
11   return ...
12 }
```

The `app` directory exposes new read-only functions to retrieve request data:

- `headers()`: Based on the Web Headers API, and can be used inside [Server Components](#) to retrieve request headers.
- `cookies()`: Based on the Web Cookies API, and can be used inside [Server Components](#) to retrieve cookies.

TS app/page.tsx

```
1 // `app` directory
2 import { cookies, headers } from 'next/headers';
3
4 async function getData() {
5   const authHeader = headers().get('authorization');
6
7   return '...';
8 }
9
10 export default async function Page() {
11   // You can use `cookies()` or `headers()` inside Server Components
12   // directly or in your data fetching function
13   const theme = cookies().get('theme');
14   const data = await getData();
15   return '...';
16 }
```

Static Site Generation (`getStaticProps`)

In the `pages` directory, the `getStaticProps` function is used to pre-render a page at build time. This function can be used to fetch data from an external API or directly from a database, and pass this data down to the entire page as it's being generated during the build.

JS pages/index.js

```
1 // `pages` directory
2
3 export async function getStaticProps() {
4   const res = await fetch(`https://...`);
5   const projects = await res.json();
6
7   return { props: { projects } };
8 }
```

```
8 }
9
10 export default function Index({ projects }) {
11   return projects.map((project) => <div>{project.name}</div>);
12 }
```

In the `app` directory, data fetching with `fetch()` will default to `cache: 'force-cache'`, which will cache the request data until manually invalidated. This is similar to `getStaticProps` in the `pages` directory.

js app/page.js

```
1 // `app` directory
2
3 // This function can be named anything
4 async function getProjects() {
5   const res = await fetch(`https://...`);
6   const projects = await res.json();
7
8   return projects;
9 }
10
11 export default async function Index() {
12   const projects = await getProjects();
13
14   return projects.map((project) => <div>{project.name}</div>);
15 }
```

Dynamic paths (`getStaticPaths`)

In the `pages` directory, the `getStaticPaths` function is used to define the dynamic paths that should be pre-rendered at build time.

js pages/posts/[id].js

```
1 // `pages` directory
2 import PostLayout from '@/components/post-layout';
3
4 export async function getStaticPaths() {
5   return {
6     paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
7   };
8 }
9
10 export async function getStaticProps({ params }) {
11   const res = await fetch(`https://.../posts/${params.id}`);
12   const post = await res.json();
13
14   return { props: { post } };
15 }
16
```

```
17 export default function Post({ post }) {
18   return <PostLayout post={post} />;
19 }
```

In the `app` directory, `getStaticPaths` is replaced with `generateStaticParams`.

`generateStaticParams` behaves similarly to `getStaticPaths`, but has a simplified API for returning route parameters and can be used inside `layouts`. The return shape of `generateStaticParams` is an array of segments instead of an array of nested `param` objects or a string of resolved paths.

Js app/posts/[id]/page.js

```
1 // `app` directory
2 import PostLayout from '@/components/post-layout';
3
4 export async function generateStaticParams() {
5   return [{ id: '1' }, { id: '2' }];
6 }
7
8 async function getPost(params) {
9   const res = await fetch(`https://.../posts/${params.id}`);
10  const post = await res.json();
11
12  return post;
13 }
14
15 export default async function Post({ params }) {
16   const post = await getPost(params);
17
18   return <PostLayout post={post} />;
19 }
```

Using the name `generateStaticParams` is more appropriate than `getStaticPaths` for the new model in the `app` directory. The `get` prefix is replaced with a more descriptive `generate`, which sits better alone now that `getStaticProps` and `getServerSideProps` are no longer necessary. The `Paths` suffix is replaced by `Params`, which is more appropriate for nested routing with multiple dynamic segments.

Replacing `fallback`

In the `pages` directory, the `fallback` property returned from `getStaticPaths` is used to define the behavior of a page that isn't pre-rendered at build time. This property can be set to `true` to show a fallback page while the page is being generated, `false` to show a 404 page, or `blocking` to generate the page at request time.

js pages/posts/[id].js

```
1 // `pages` directory
2
3 export async function getStaticPaths() {
4   return {
5     paths: [],
6     fallback: 'blocking'
7   };
8 }
9
10 export async function getStaticProps({ params }) {
11   ...
12 }
13
14 export default function Post({ post }) {
15   return ...
16 }
```

In the `app` directory the `config.dynamicParams` property controls how params outside of `generateStaticParams` are handled:

- `true`: (default) Dynamic segments not included in `generateStaticParams` are generated on demand.
- `false`: Dynamic segments not included in `generateStaticParams` will return a 404.

This replaces the `fallback: true | false | 'blocking'` option of `getStaticPaths` in the `pages` directory. The `fallback: 'blocking'` option is not included in `dynamicParams` because the difference between `'blocking'` and `true` is negligible with streaming.

js app/posts/[id]/page.js

```
1 // `app` directory
2
3 export const dynamicParams = true;
4
5 export async function generateStaticParams() {
6   return [...]
7 }
8
9 async function getPost(params) {
10   ...
11 }
12
13 export default async function Post({ params }) {
14   const post = await getPost(params);
15
16   return ...
17 }
```

With `dynamicParams` set to `true` (the default), when a route segment is requested that hasn't been generated, it will be server-rendered and cached as `static data` on success.

Incremental Static Regeneration (`getStaticProps` with `revalidate`)

In the `pages` directory, the `getStaticProps` function allows you to add a `revalidate` field to automatically regenerate a page after a certain amount of time. This is called [Incremental Static Regeneration \(ISR\)](#) and helps you update static content without redeploying.

JS pages/index.js

```
1 // `pages` directory
2
3 export async function getStaticProps() {
4   const res = await fetch(`https://.../posts`);
5   const posts = await res.json();
6
7   return {
8     props: { posts },
9     revalidate: 60,
10    };
11 }
12
13 export default function Index({ posts }) {
14   return (
15     <Layout>
16       <PostList posts={posts} />
17     </Layout>
18   );
19 }
```

In the `app` directory, data fetching with `fetch()` can use `revalidate`, which will cache the request for the specified amount of seconds.

JS app/page.js

```
1 // `app` directory
2
3 async function getPosts() {
4   const res = await fetch(`https://.../posts`, { next: { revalidate: 60 } });
5   const data = await res.json();
6
7   return data.posts;
8 }
9
10 export default async function PostList() {
11   const posts = await getPosts();
12
13   return posts.map((post) => <div>{post.name}</div>);
```

API Routes

API Routes continue to work in the `pages/api` directory without any changes. However, they have been replaced by [Route Handlers](#) in the `app` directory.

Route Handlers allow you to create custom request handlers for a given route using the [Web Request ↗](#) and [Response ↗](#) APIs.

 app/api/route.ts

```
export async function GET(request: Request) {}
```

Note: If you previously used API routes to call an external API from the client, you can now use [Server Components](#) instead to securely fetch data. Learn more about [data fetching](#).

Step 7: Styling

In the `pages` directory, global stylesheets are restricted to only `pages/_app.js`. With the `app` directory, this restriction has been lifted. Global styles can be added to any layout, page, or component.

- [CSS Modules](#)
- [Tailwind CSS](#)
- [Global Styles](#)
- [CSS-in-JS](#)
- [External Stylesheets](#)
- [Sass](#)

Tailwind CSS

If you're using Tailwind CSS, you'll need to add the `app` directory to your `tailwind.config.js` file:

 tailwind.config.js

```
1 module.exports = {
2   content: [
3     './app/**/*.{js,ts,jsx,tsx,mdx}', // <-- Add this line
4     './pages/**/*.{js,ts,jsx,tsx,mdx}',
5     './components/**/*.{js,ts,jsx,tsx,mdx}',
6   ],
7 };
```

You'll also need to import your global styles in your `app/layout.js` file:

js app/layout.js

```
1 import '../styles/globals.css';
2
3 export default function RootLayout({ children }) {
4   return (
5     <html lang="en">
6       <body>{children}</body>
7     </html>
8   );
9 }
```

Learn more about [styling with Tailwind CSS](#)

Codemods

Next.js provides Codemod transformations to help upgrade your codebase when a feature is deprecated. See [Codemods](#) for more information.

> Menu

Pages Router > ... > Upgrading > Codemods

Codemods

Codemods are transformations that run on your codebase programmatically. This allows a large number of changes to be programmatically applied without having to manually go through every file.

Next.js provides Codemod transformations to help upgrade your Next.js codebase when an API is updated or deprecated.

Usage

In your terminal, navigate (`cd`) into your project's folder, then run:

> Terminal

```
npx @next/codemod <transform> <path>
```

Replacing `<transform>` and `<path>` with appropriate values.

- `transform` - name of transform
- `path` - files or directory to transform
- `--dry` Do a dry-run, no code will be edited
- `--print` Prints the changed output for comparison

Next.js Codemods

13.2

Use Built-in Font

built-in-next-font

>_ Terminal



```
npx @next/codemod@latest built-in-next-font
```

This codemod uninstalls the `@next/font` package and transforms `@next/font` imports into the built-in `next/font`.

For example:

```
import { Inter } from '@next/font/google';
```

Transforms into:

```
import { Inter } from 'next/font/google';
```

13.0

Rename Next Image Imports

next-image-to-legacy-image

>_ Terminal



```
npx @next/codemod@latest next-image-to-legacy-image ./pages
```

Safely renames `next/image` imports in existing Next.js 10, 11, or 12 applications to `next/legacy/image` in Next.js 13. Also renames `next/future/image` to `next/image`.

For example:

js pages/index.js



```
1 import Image1 from 'next/image';
2 import Image2 from 'next/future/image';
3
4 export default function Home() {
5   return (
6     <div>
7       <Image1 src="/test.jpg" width="200" height="300" />
8       <Image2 src="/test.png" width="500" height="400" />
9     </div>
```

```
10  );
11 }
```

Transforms into:

js pages/index.js

```
1 // 'next/image' becomes 'next/legacy/image'
2 import Image1 from 'next/legacy/image';
3 // 'next/future/image' becomes 'next/image'
4 import Image2 from 'next/image';
5
6 export default function Home() {
7   return (
8     <div>
9       <Image1 src="/test.jpg" width="200" height="300" />
10      <Image2 src="/test.png" width="500" height="400" />
11    </div>
12  );
13 }
```

Migrate to the New Image Component

next-image-experimental

>_ Terminal

```
npx @next/codemod@latest next-image-experimental ./pages
```

Dangerously migrates from `next/legacy/image` to the new `next/image` by adding inline styles and removing unused props.

- Removes `layout` prop and adds `style`.
- Removes `objectFit` prop and adds `style`.
- Removes `objectPosition` prop and adds `style`.
- Removes `lazyBoundary` prop.
- Removes `lazyRoot` prop.

Remove `<a>` Tags From Link Components

new-link

>_ Terminal

```
npx @next/codemod@latest new-link ./pages
```

Remove `<a>` tags inside [Link Components](#), or add a `legacyBehavior` prop to Links that cannot be auto-fixed.

For example:

```
1 <Link href="/about">
2   <a>About</a>
3 </Link>
4 // transforms into
5 <Link href="/about">
6   About
7 </Link>
8
9 <Link href="/about">
10  <a onClick={() => console.log('clicked')}>About</a>
11 </Link>
12 // transforms into
13 <Link href="/about" onClick={() => console.log('clicked')}>
14   About
15 </Link>
```

In cases where auto-fixing can't be applied, the `legacyBehavior` prop is added. This allows your app to keep functioning using the old behavior for that particular link.

```
1 const Component = () => <a>About</a>
2
3 <Link href="/about">
4   <Component />
5 </Link>
6 // becomes
7 <Link href="/about" legacyBehavior>
8   <Component />
9 </Link>
```

11

Migrate from CRA

cra-to-next

_ Terminal

```
npx @next/codemod cra-to-next
```

Migrates a Create React App project to Next.js; creating a Pages Router and necessary config to match behavior. Client-side only rendering is leveraged initially to prevent breaking compatibility due to `window` usage during SSR and can be enabled seamlessly to allow the gradual adoption of Next.js specific features.

Please share any feedback related to this transform [in this discussion ↗](#).

10

Add React imports

`add-missing-react-import`

>_ Terminal

```
npx @next/codemod add-missing-react-import
```

Transforms files that do not import `React` to include the import in order for the new [React JSX transform ↗](#) to work.

For example:

js my-component.js

```
1 export default class Home extends React.Component {
2   render() {
3     return <div>Hello World</div>;
4   }
5 }
```

Transforms into:

js my-component.js

```
1 import React from 'react';
2 export default class Home extends React.Component {
3   render() {
4     return <div>Hello World</div>;
5   }
6 }
```

9

Transform Anonymous Components into Named Components

`name-default-component`

>_ Terminal



```
npx @next/codemod name-default-component
```

Versions 9 and above.

Transforms anonymous components into named components to make sure they work with [Fast Refresh ↗](#).

For example:

JS my-component.js



```
1 export default function () {
2   return <div>Hello World</div>;
3 }
```

Transforms into:

JS my-component.js



```
1 export default function MyComponent() {
2   return <div>Hello World</div>;
3 }
```

The component will have a camel-cased name based on the name of the file, and it also works with arrow functions.

8

Transform AMP HOC into page config

`withamp-to-config`

>_ Terminal



```
npx @next/codemod withamp-to-config
```

Transforms the `withAmp` HOC into Next.js 9 page configuration.

For example:

```
1 // Before
```

```
2 import { withAmp } from 'next/amp';
3
4 function Home() {
5   return <h1>My AMP Page</h1>;
6 }
7
8 export default withAmp(Home);
```

```
1 // After
2 export default function Home() {
3   return <h1>My AMP Page</h1>;
4 }
5
6 export const config = {
7   amp: true,
8 };
```

6

Use `withRouter`

`url-to-withrouter`

>_ Terminal

```
npx @next/codemod url-to-withrouter
```

Transforms the deprecated automatically injected `url` property on top level pages to using `withRouter` and the `router` property it injects. Read more here: <https://nextjs.org/docs/messages/url-deprecated>

For example:

From

```
1 import React from 'react';
2 export default class extends React.Component {
3   render() {
4     const { pathname } = this.props.url;
5     return <div>Current pathname: {pathname}</div>;
6   }
7 }
```

To

```
1 import React from 'react';
2 import { withRouter } from 'next/router';
```

```
3 export default withRouter(  
4   class extends React.Component {  
5     render() {  
6       const { pathname } = this.props.router;  
7       return <div>Current pathname: {pathname}</div>;  
8     }  
9   },  
10 );
```

This is one case. All the cases that are transformed (and tested) can be found in the [__testfixtures directory ↗](#).

> Menu

[Pages Router](#) > ... > [Upgrading](#) > [Version 10](#)

Version 10

There were no breaking changes between versions 9 and 10.

To upgrade to version 10, run the following command:

>_ Terminal

```
1 npm install next@10
2
3 yarn add next@10
```


> Menu

Pages Router > ... > Upgrading > Version 11

Version 11

To upgrade to version 11, run the following command:

> Terminal

```
1 npm install next@11
2
3 yarn add next@11
```



Webpack 5

Webpack 5 is now the default for all Next.js applications. If you did not have a custom webpack configuration, your application is already using webpack 5. If you do have a custom webpack configuration, you can refer to the [Next.js webpack 5 documentation](#) for upgrade guidance.

Cleaning the `distDir` is now a default

The build output directory (defaults to `.next`) is now cleared by default except for the Next.js caches. You can refer to [the cleaning `distDir` RFC](#) for more information.

If your application was relying on this behavior previously you can disable the new default behavior by adding the `cleanDistDir: false` flag in `next.config.js`.

`PORT` is now supported for `next dev` and `next start`

Next.js 11 supports the `PORT` environment variable to set the port the application runs on. Using `-p` / `--port` is still recommended but if you were prohibited from using `-p` in any way you can now use `PORT` as an alternative:

Example:

```
PORT=4000 next start
```

next.config.js customization to import images

Next.js 11 supports static image imports with `next/image`. This new feature relies on being able to process image imports. If you previously added the `next-images` or `next-optimized-images` packages you can either move to the new built-in support using `next/image` or disable the feature:

JS next.config.js

```
1 module.exports = {  
2   images: {  
3     disableStaticImages: true,  
4   },  
5 };
```

Remove `super.componentDidCatch()` from `pages/_app.js`

The `next/app` component's `componentDidCatch` was deprecated in Next.js 9 as it's no longer needed and has since been a no-op. In Next.js 11, it was removed.

If your `pages/_app.js` has a custom `componentDidCatch` method you can remove `super.componentDidCatch` as it is no longer needed.

Remove `Container` from `pages/_app.js`

This export was deprecated in Next.js 9 as it's no longer needed and has since been a no-op with a warning during development. In Next.js 11 it was removed.

If your `pages/_app.js` imports `Container` from `next/app` you can remove `Container` as it was removed. Learn more in [the documentation](#).

Remove `props.url` usage from page components

This property was deprecated in Next.js 4 and has since shown a warning during development. With the introduction of `getStaticProps` / `getServerSideProps` these methods already disallowed the usage of `props.url`. In Next.js 11, it was removed completely.

You can learn more in [the documentation](#).

Remove `unsized` property on `next/image`

The `unsized` property on `next/image` was deprecated in Next.js 10.0.1. You can use `layout="fill"` instead. In Next.js 11 `unsized` was removed.

Remove `modules` property on `next/dynamic`

The `modules` and `render` option for `next/dynamic` were deprecated in Next.js 9.5. This was done in order to make the `next/dynamic` API closer to `React.lazy`. In Next.js 11, the `modules` and `render` options were removed.

This option hasn't been mentioned in the documentation since Next.js 8 so it's less likely that your application is using it.

If your application does use `modules` and `render` you can refer to [the documentation](#).

Remove `Head.rewind`

`Head.rewind` has been a no-op since Next.js 9.5, in Next.js 11 it was removed. You can safely remove your usage of `Head.rewind`.

Moment.js locales excluded by default

Moment.js includes translations for a lot of locales by default. Next.js now automatically excludes these locales by default to optimize bundle size for applications using Moment.js.

To load a specific locale use this snippet:

```
1 import moment from 'moment';
2 import 'moment/locale/ja';
3
4 moment.locale('ja');
```

You can opt-out of this new default by adding `excludeDefaultMomentLocales: false` to `next.config.js` if you do not want the new behavior, do note it's highly recommended to not disable this new optimization as it significantly reduces the size of Moment.js.

Update usage of `router.events`

In case you're accessing `router.events` during rendering, in Next.js 11 `router.events` is no longer provided during pre-rendering. Ensure you're accessing `router.events` in `useEffect`:

```
1 useEffect(() => {
2   const handleRouteChange = (url, { shallow }) => {
3     console.log(
4       `App is changing to ${url} ${
5         shallow ? 'with' : 'without'
6       } shallow routing`,
7     );
}
```

```
8  };
9
10 router.events.on('routeChangeStart', handleRouteChange);
11
12 // If the component is unmounted, unsubscribe
13 // from the event with the `off` method:
14 return () => {
15   router.events.off('routeChangeStart', handleRouteChange);
16 };
17 }, [router]);
```

If your application uses `router.router.events` which was an internal property that was not public please make sure to use `router.events` as well.

React 16 to 17

React 17 introduced a new [JSX Transform](#) that brings a long-time Next.js feature to the wider React ecosystem: Not having to `import React from 'react'` when using JSX. When using React 17 Next.js will automatically use the new transform. This transform does not make the `React` variable global, which was an unintended side-effect of the previous Next.js implementation. A [codemod is available](#) to automatically fix cases where you accidentally used `React` without importing it.

Most applications already use the latest version of React, with Next.js 11 the minimum React version has been updated to 17.0.2.

To upgrade you can run the following command:

```
npm install react@latest react-dom@latest
```

Or using `yarn`:

```
yarn add react@latest react-dom@latest
```

> Menu

Pages Router > ... > Upgrading > Version 12

Version 12

To upgrade to version 12, run the following command:

>_ Terminal



```
1 npm install next@12
2
3 yarn add next@12
4
5 pnpm update next@12
```

Upgrading to 12.2

[Middleware](#) - If you were using Middleware prior to 12.2, please see the [upgrade guide](#) for more information.

Upgrading to 12.0

[Minimum Node.js Version ↗](#) - The minimum Node.js version has been bumped from 12.0.0 to 12.22.0 which is the first version of Node.js with native ES Modules support.

[Minimum React Version ↗](#) - The minimum required React version is 17.0.2. To upgrade you can run the following command in the terminal:

>_ Terminal



```
1 npm install react@latest react-dom@latest
2
3 yarn add react@latest react-dom@latest
4
5 pnpm update react@latest react-dom@latest
```

SWC replacing Babel

Next.js now uses the Rust-based compiler [SWC ↗](#) to compile JavaScript/TypeScript. This new compiler is up to 17x faster than Babel when compiling individual files and up to 5x faster Fast Refresh.

Next.js provides full backward compatibility with applications that have [custom Babel configuration](#). All transformations that Next.js handles by default like styled-jsx and tree-shaking of `getStaticProps` / `getStaticPaths` / `getServerSideProps` have been ported to Rust.

When an application has a custom Babel configuration, Next.js will automatically opt-out of using SWC for compiling JavaScript/Typescript and will fall back to using Babel in the same way that it was used in Next.js 11.

Many of the integrations with external libraries that currently require custom Babel transformations will be ported to Rust-based SWC transforms in the near future. These include but are not limited to:

- Styled Components
- Emotion
- Relay

In order to prioritize transforms that will help you adopt SWC, please provide your `.babelrc` on [this feedback thread ↗](#).

SWC replacing Terser for minification

You can opt-in to replacing Terser with SWC for minifying JavaScript up to 7x faster using a flag in `next.config.js`:

`JS` `next.config.js`

```
1 module.exports = {
2   swcMinify: true,
3 };
```

Minification using SWC is an opt-in flag to ensure it can be tested against more real-world Next.js applications before it becomes the default in Next.js 12.1. If you have feedback about minification, please leave it on [this feedback thread ↗](#).

Improvements to styled-jsx CSS parsing

On top of the Rust-based compiler we've implemented a new CSS parser based on the one used for the styled-jsx Babel transform. This new parser has improved handling of CSS and now errors when invalid CSS is used that would previously slip through and cause unexpected behavior.

Because of this change invalid CSS will throw an error during development and `next build`. This change only affects styled-jsx usage.

`next/image` changed wrapping element

`next/image` now renders the `` inside a `` instead of `<div>`.

If your application has specific CSS targeting span such as `.container span`, upgrading to Next.js 12 might incorrectly match the wrapping element inside the `<Image>` component. You can avoid this by restricting the selector to a specific class such as `.container span.item` and updating the relevant component with that className, such as ``.

If your application has specific CSS targeting the `next/image <div>` tag, for example `.container div`, it may not match anymore. You can update the selector `.container span`, or preferably, add a new `<div className="wrapper">` wrapping the `<Image>` component and target that instead such as `.container .wrapper`.

The `className` prop is unchanged and will still be passed to the underlying `` element.

See the [documentation](#) for more info.

HMR connection now uses a WebSocket

Previously, Next.js used a [server-sent events ↗](#) connection to receive HMR events. Next.js 12 now uses a WebSocket connection.

In some cases when proxying requests to the Next.js dev server, you will need to ensure the upgrade request is handled correctly. For example, in `nginx` you would need to add the following configuration:

```
1 location /_next/webpack-hmr {
2     proxy_pass http://localhost:3000/_next/webpack-hmr;
3     proxy_http_version 1.1;
4     proxy_set_header Upgrade $http_upgrade;
5     proxy_set_header Connection "upgrade";
6 }
```

If you are using Apache (2.x), you can add the following configuration to enable web sockets to the server. Review the port, host name and server names.

```
<VirtualHost *:443>
# ServerName yourwebsite.local
ServerName "${WEBSITE_SERVER_NAME}"
ProxyPass / http://localhost:3000/
ProxyPassReverse / http://localhost:3000/
# Next.js 12 uses websocket
<Location /_next/webpack-hmr>
```

```
RewriteEngine On
RewriteCond %{QUERY_STRING} transport=websocket [NC]
RewriteCond %{HTTP:Upgrade} websocket [NC]
RewriteCond %{HTTP:Connection} upgrade [NC]
RewriteRule /(.*) ws://localhost:3000/_next/webpack-hmr/$1 [P,L]
ProxyPass ws://localhost:3000/_next/webpack-hmr retry=0 timeout=30
ProxyPassReverse ws://localhost:3000/_next/webpack-hmr
</Location>
</VirtualHost>
```

For custom servers, such as `express`, you may need to use `app.all` to ensure the request is passed correctly, for example:

```
1 app.all('/_next/webpack-hmr', (req, res) => {
2   nextjsRequestHandler(req, res);
3 });
```

Webpack 4 support has been removed

If you are already using webpack 5 you can skip this section.

Next.js has adopted webpack 5 as the default for compilation in Next.js 11. As communicated in the [webpack 5 upgrading documentation](#) Next.js 12 removes support for webpack 4.

If your application is still using webpack 4 using the opt-out flag, you will now see an error linking to the [webpack 5 upgrading documentation](#).

`target` option deprecated

If you do not have `target` in `next.config.js` you can skip this section.

The target option has been deprecated in favor of built-in support for tracing what dependencies are needed to run a page.

During `next build`, Next.js will automatically trace each page and its dependencies to determine all of the files that are needed for deploying a production version of your application.

If you are currently using the `target` option set to `serverless`, please read the [documentation on how to leverage the new output](#).

> Menu

Pages Router > ... > Upgrading > Version 13

Version 13

Upgrading from 12 to 13

To update to Next.js version 13, run the following command using your preferred package manager:

> Terminal



```
1 npm i next@latest react@latest react-dom@latest eslint-config-next@latest
2 # or
3 yarn add next@latest react@latest react-dom@latest eslint-config-next@latest
4 # or
5 pnpm up next react react-dom eslint-config-next --latest
```

v13 Summary

- The [Supported Browsers](#) have been changed to drop Internet Explorer and target modern browsers.
- The minimum Node.js version has been bumped from 12.22.0 to 14.18.0, since 12.x has reached end-of-life.
- The minimum React version has been bumped from 17.0.2 to 18.2.0.
- The `swcMinify` configuration property was changed from `false` to `true`. See [Next.js Compiler](#) for more info.
- The `next/image` import was renamed to `next/legacy/image`. The `next/future/image` import was renamed to `next/image`. A [codemod is available](#) to safely and automatically rename your imports.
- The `next/link` child can no longer be `<a>`. Add the `legacyBehavior` prop to use the legacy behavior or remove the `<a>` to upgrade. A [codemod is available](#) to automatically upgrade your code.
- The `target` configuration property has been removed and superseded by [Output File Tracing](#).

Migrating shared features

Next.js 13 introduces a new `app` directory with new features and conventions. However, upgrading to Next.js 13 does **not** require using the new `app` directory.

You can continue using `pages` with new features that work in both directories, such as the updated [Image component](#), [Link component](#), [Script component](#), and [Font optimization](#).

<Image/> Component

Next.js 12 introduced many improvements to the Image Component with a temporary import:

`next/future/image`. These improvements included less client-side JavaScript, easier ways to extend and style images, better accessibility, and native browser lazy loading.

Starting in Next.js 13, this new behavior is now the default for `next/image`.

There are two codemods to help you migrate to the new Image Component:

- [next-image-to-legacy-image](#): This codemod will safely and automatically rename `next/image` imports to `next/legacy/image` to maintain the same behavior as Next.js 12. We recommend running this codemod to quickly update to Next.js 13 automatically.
- [next-image-experimental](#): After running the previous codemod, you can optionally run this experimental codemod to upgrade `next/legacy/image` to the new `next/image`, which will remove unused props and add inline styles. Please note this codemod is experimental and only covers static usage (such as `<Image src={img} layout="responsive" />`) but not dynamic usage (such as `<Image {...props} />`).

Alternatively, you can manually update by following the [migration guide](#) and also see the [legacy comparison](#).

<Link> Component

The [<Link> Component](#) no longer requires manually adding an `<a>` tag as a child. This behavior was added as an experimental option in [version 12.2 ↗](#) and is now the default. In Next.js 13, `<Link>` always renders `<a>` and allows you to forward props to the underlying tag.

For example:

```
1 import Link from 'next/link'  
2  
3 // Next.js 12: `<a>` has to be nested otherwise it's excluded
```

```
4 <Link href="/about">
5   <a>About</a>
6 </Link>
7
8 // Next.js 13: `<Link>` always renders `<a>` under the hood
9 <Link href="/about">
10  About
11 </Link>
```

To upgrade your links to Next.js 13, you can use the [new-link codemod](#).

<Script> Component

The behavior of [next/script](#) has been updated to support both [pages](#) and [app](#). If incrementally adopting [app](#), read the [upgrade guide](#).

Font Optimization

Previously, Next.js helped you optimize fonts by inlining font CSS. Version 13 introduces the new [next/font](#) module which gives you the ability to customize your font loading experience while still ensuring great performance and privacy.

See [Optimizing Fonts](#) to learn how to use [next/font](#).

> Menu

Pages Router > ... > Upgrading > Version 9

Upgrading to Version 9

To upgrade to version 9, run the following command:

> Terminal

```
1 npm install next@9
2
3 yarn add next@9
```



Production Deployment on Vercel

If you previously configured `routes` in your `vercel.json` file for dynamic routes, these rules can be removed when leveraging Next.js 9's new [Dynamic Routing feature](#).

Next.js 9's dynamic routes are **automatically configured on Vercel** and do not require any `vercel.json` customization.

You can read more about [Dynamic Routing here](#).

Check your Custom App File (`pages/_app.js`)

If you previously copied the [Custom <App>](#) example, you may be able to remove your `getInitialProps`.

Removing `getInitialProps` from `pages/_app.js` (when possible) is important to leverage new Next.js features!

The following `getInitialProps` does nothing and may be removed:

```
1 class MyApp extends App {
2   // Remove me, I do nothing!
3   static async getInitialProps({ Component, ctx }) {
4     let pageProps = {};
5
6     if (Component.getInitialProps) {
7       pageProps = await Component.getInitialProps(ctx);
8     }
9
10    return { pageProps };
11  }
12
13  render() {
14    // ... etc
15  }
16}
```

Breaking Changes

`@zeit/next-typescript` is no longer necessary

Next.js will now ignore usage `@zeit/next-typescript` and warn you to remove it. Please remove this plugin from your `next.config.js`.

Remove references to `@zeit/next-typescript/babel` from your custom `.babelrc` (if present).

The usage of `fork-ts-checker-webpack-plugin` [↗](#) should also be removed from your `next.config.js`.

TypeScript Definitions are published with the `next` package, so you need to uninstall `@types/next` as they would conflict.

The following types are different:

This list was created by the community to help you upgrade, if you find other differences please send a pull-request to this list to help other users.

From:

```
1 import { NextContext } from 'next';
2 import { NextAppContext, DefaultAppIProps } from 'next/app';
3 import { NextDocumentContext, DefaultDocumentIProps } from 'next/document';
```

to

```
1 import { NextPageContext } from 'next';
2 import { ApplicationContext, AppInitialProps } from 'next/app';
3 import { DocumentContext, DocumentInitialProps } from 'next/document';
```

The `config` key is now an export on a page

You may no longer export a custom variable named `config` from a page (i.e. `export { config }` / `export const config ...`). This exported variable is now used to specify page-level Next.js configuration like Opt-in AMP and API Route features.

You must rename a non-Next.js-purposed `config` export to something different.

`next/dynamic` no longer renders "loading..." by default while loading

Dynamic components will not render anything by default while loading. You can still customize this behavior by setting the `loading` property:

```
1 import dynamic from 'next/dynamic';
2
3 const DynamicComponentWithCustomLoading = dynamic(
4   () => import('../components/hello2'),
5   {
6     loading: () => <p>Loading</p>,
7   },
8 );
```

`withAmp` has been removed in favor of an exported configuration object

Next.js now has the concept of page-level configuration, so the `withAmp` higher-order component has been removed for consistency.

This change can be **automatically migrated by running the following commands in the root of your Next.js project:**

>_ Terminal



```
curl -L https://github.com/vercel/next-codemod/archive/master.tar.gz | tar -xz --strip=2 next
```

To perform this migration by hand, or view what the codemod will produce, see below:

Before

```
1 import { withAmp } from 'next/amp'  
2  
3 function Home() {  
4   return <h1>My AMP Page</h1>  
5 }  
6  
7 export default withAmp(Home)  
8 // or  
9 export default withAmp(Home, { hybrid: true })
```

After

```
1 export default function Home() {  
2   return <h1>My AMP Page</h1>;  
3 }  
4  
5 export const config = {  
6   amp: true,  
7   // or  
8   amp: 'hybrid',  
9 };
```

next export no longer exports pages as index.html

Previously, exporting `pages/about.js` would result in `out/about/index.html`. This behavior has been changed to result in `out/about.html`.

You can revert to the previous behavior by creating a `next.config.js` with the following content:

`js` next.config.js

```
1 module.exports = {  
2   trailingSlash: true,  
3 };
```

pages/api/ is treated differently

Pages in `pages/api/` are now considered [API Routes ↗](#). Pages in this directory will no longer contain a client-side bundle.

Deprecated Features

next/dynamic has deprecated loading multiple modules at once

The ability to load multiple modules at once has been deprecated in `next/dynamic` to be closer to React's implementation (`React.lazy` and `Suspense`).

Updating code that relies on this behavior is relatively straightforward! We've provided an example of a before/after to help you migrate your application:

Before

```
1 import dynamic from 'next/dynamic';
2
3 const HelloBundle = dynamic({
4   modules: () => {
5     const components = {
6       Hello1: () => import('../components/hello1').then((m) => m.default),
7       Hello2: () => import('../components/hello2').then((m) => m.default),
8     };
9
10    return components;
11  },
12  render: (props, { Hello1, Hello2 }) => (
13    <div>
14      <h1>{props.title}</h1>
15      <Hello1 />
16      <Hello2 />
17    </div>
18  ),
19});
20
21 function DynamicBundle() {
22   return <HelloBundle title="Dynamic Bundle" />;
23 }
24
25 export default DynamicBundle;
```

After

```
1 import dynamic from 'next/dynamic';
2
3 const Hello1 = dynamic(() => import('../components/hello1'));
4 const Hello2 = dynamic(() => import('../components/hello2'));
5
6 function HelloBundle({ title }) {
7   return (
8     <div>
```

```
9     <h1>{title}</h1>
10    <Hello1 />
11    <Hello2 />
12  </div>
13 );
14 }
15
16 function DynamicBundle() {
17   return <HelloBundle title="Dynamic Bundle" />;
18 }
19
20 export default DynamicBundle;
```