

> Menu

App Router

The App Router is a new paradigm for building applications using React's latest features. If you're already familiar with Next.js, you'll find that the App Router is a natural evolution of the existing file-system based router in the [Pages Router](#).

For new applications, we recommend using the App Router. For existing applications, you can [incrementally migrate to the App Router](#).

This section of the documentation includes the features available in the App Router:

[Building Your Application](#)

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

[API Reference](#)

Next.js API Reference for the App Router.

> Menu

App Router > API Reference

API Reference

The Next.js API reference is divided into the following sections:

Components

API Reference for Next.js built-in components.

File Conventions

API Reference for Next.js Special Files.

Functions

API Reference for Next.js Functions and Hooks.

next.config.js Options

Learn how to configure your application with next.config.js.

create-next-app

Create Next.js apps in one command with create-next-app.

Edge Runtime

API Reference for the Edge Runtime.

[Next.js CLI](#)

The Next.js CLI allows you to start, build, and export your application. Learn more about it [here](#).

> Menu

App Router > API Reference > Components

Components

Font

Optimizing loading web fonts with the built-in `next/font` loaders.

<Image>

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

<Link>

Enable fast client-side navigation with the built-in `next/link` component.

<Script>

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

> Menu

App 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

App Router > ... > Components > <Image>

<Image>

► Examples

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
width	width={500}	Integer (px)	Yes
height	height={500}	Integer (px)	Yes

Prop	Example	Type	Required
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  );

```

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

App 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.

TS app/page.tsx

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

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 }
```

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 `app/blog/[slug]/page.js`:

JS app/blog/page.js

```
1 import Link from 'next/link';
2
3 function Page({ 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}
```

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:

```
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 }
```

> Menu

App 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 Script from 'next/script';
2
3 export default function RootLayout({
4   children,
5 }: {
6   children: React.ReactNode;
7 }) {
8   return (
9     <html lang="en">
10       <body>{children}</body>
11       <Script
12         src="https://example.com/script.js"
13         strategy="beforeInteractive"
14       />
15     </html>
16   );
17 }
```

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.

```
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   );
9 }
```

```
8    );
9 }
```

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`:

JS next.config.js

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

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

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 }
```

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.

TS app/page.tsx

```
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:

app/page.tsx

```
1 'use client';
2
3 import { useRef } from 'react';
4 import Script from 'next/script';
5
6 export default function Page() {
7   const mapRef = useRef();
8
9   return (
10     <>
11       <div ref={mapRef}></div>
12       <Script
13         id="google-maps"
14         src="https://maps.googleapis.com/maps/api/js"
15         onReady={() => {
16           new google.maps.Map(mapRef.current, {
17             center: { lat: -34.397, lng: 150.644 },
18             zoom: 8,
19           });
20         }}
21       />
22     </>
23   );
24 }
```

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:

app/page.tsx

```
1  'use client';
2
3  import Script from 'next/script';
4
5  export default function Page() {
6    return (
7      <>
8        <Script
9          src="https://example.com/script.js"
10         onError={(e: Error) => {
11           console.error('Script failed to load', e);
12         }}
13       />
14     </>
15   );
16 }
```

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

App 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

App 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

App Router > API Reference > File Conventions

File Conventions

[default.js](#)

API Reference for the default.js file.

[error.js](#)

API reference for the error.js special file.

[layout.js](#)

API reference for the layout.js file.

[loading.js](#)

API reference for the loading.js file.

[not-found.js](#)

API reference for the not-found.js file.

[page.js](#)

API reference for the page.js file.

[route.js](#)

API reference for the route.js special file.

[Route Segment Config](#)

Learn about how to configure options for Next.js route segments.

[template.js](#)

API Reference for the template.js file.

[Metadata Files](#)

API documentation for the metadata file conventions.

> Menu

App Router > ... > File Conventions > default.js

default.js

This documentation is still being written. Please check back later.

> Menu

App Router > ... > File Conventions > error.js

error.js

An **error** file defines an error UI boundary for a route segment.

A screenshot of a code editor window titled "app/dashboard/error.tsx". The code is written in TypeScript and defines an "Error" component. It uses the "useEffect" hook to log errors to the console and includes a button to attempt a re-render.

```
1 'use client'; // Error components must be Client Components
2
3 import { useEffect } from 'react';
4
5 export default function Error({
6   error,
7   reset,
8 }: {
9   error: Error;
10  reset: () => void;
11}) {
12  useEffect(() => {
13    // Log the error to an error reporting service
14    console.error(error);
15  }, [error]);
16
17  return (
18    <div>
19      <h2>Something went wrong!</h2>
20      <button
21        onClick={
22          // Attempt to recover by trying to re-render the segment
23          () => reset()
24        }
25      >
26        Try again
27      </button>
28    </div>
29  );
30}
```

⌄



Props

error

An instance of an [Error](#) object. This error can happen on the server or the client.

reset

A function to reset the error boundary, which does not return a response.

Good to know:

- `error.js` boundaries must be [Client Components](#).
- An `error.js` boundary will **not** handle errors thrown in a `layout.js` component in the **same** segment because the error boundary is nested **inside** that layouts component.
 - To handle errors for a specific layout, place an `error.js` file in the layouts parent segment.
 - To handle errors within the root layout or template, use a variation of `error.js` called `app/global-error.js`.

global-error.js

To specifically handle errors in root `layout.js`, use a variation of `error.js` called `app/global-error.js` located in the root `app` directory.

ts app/global-error.tsx

```
1 'use client';
2
3 export default function GlobalError({
4   error,
5   reset,
6 }: {
7   error: Error;
8   reset: () => void;
9 }) {
10   return (
11     <html>
12       <body>
13         <h2>Something went wrong!</h2>
14         <button onClick={() => reset()}>Try again</button>
15       </body>
16     </html>
```

```
17    );
18 }
```

Good to know:

- `global-error.js` replaces the root `layout.js` when active and so **must** define its own `<html>` and `<body>` tags.
- While designing error UI, you may find it helpful to use the [React Developer Tools](#) ↗ to manually toggle Error boundaries.

> Menu

App Router > ... > File Conventions > layout.js

layout.js

A **layout** is UI that is shared between routes.

TS app/dashboard/layout.tsx

```
1 export default function DashboardLayout({  
2   children,  
3 }: {  
4   children: React.ReactNode;  
5 }) {  
6   return <section>{children}</section>;  
7 }
```

A **root layout** is the top-most layout in the root `app` directory. It is used to define the `<html>` and `<body>` tags and other globally shared UI.

TS app/layout.tsx

```
1 export default function RootLayout({  
2   children,  
3 }: {  
4   children: React.ReactNode;  
5 }) {  
6   return (  
7     <html lang="en">  
8       <body>{children}</body>  
9     </html>  
10    );  
11 }
```

Props

children (required)

Layout components should accept and use a `children` prop. During rendering, `children` will be populated with the route segments the layout is wrapping. These will primarily be the component of a child [Layout](#) (if it exists) or [Page](#), but could also be other special files like [Loading](#) or [Error](#) when applicable.

params (optional)

The [dynamic route parameters](#) object from the root segment down to that layout.

Example	URL	params
<code>app/dashboard/[team]/layout.js</code>	<code>/dashboard/1</code>	<code>{ team: '1' }</code>
<code>app/shop/[tag]/[item]/layout.js</code>	<code>/shop/1/2</code>	<code>{ tag: '1', item: '2' }</code>
<code>app/blog/[...slug]/layout.js</code>	<code>/blog/1/2</code>	<code>{ slug: ['1', '2'] }</code>

For example:

```
ts app/shop/[tag]/[item]/layout.ts
```

```
1 export default function ShopLayout({
2   children,
3   params,
4 }: {
5   children: React.ReactNode;
6   params: {
7     tag: string;
8     item: string;
9   };
10 }) {
11   // URL -> /shop/shoes/nike-air-max-97
12   // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
13   return <section>{children}</section>;
14 }
```

Good to know

Layout's do not receive `searchParams`

Unlike [Pages](#), Layout components **do not** receive the `searchParams` prop. This is because a shared layout is [not re-rendered during navigation](#) which could lead to stale `searchParams` between navigations.

When using client-side navigation, Next.js automatically only renders the part of the page below the common layout between two routes.

For example, in the following directory structure, `dashboard/layout.tsx` is the common layout for both `/dashboard/settings` and `/dashboard/analytics`:

```
1 app
2   └── dashboard
3     ├── layout.tsx
4     ├── settings
5     |   └── page.tsx
6     └── analytics
7       └── page.js
```

When navigating from `/dashboard/settings` to `/dashboard/analytics`, `page.tsx` in `/dashboard/analytics` will be rendered on the server because it is UI that changed, while `dashboard/layout.tsx` will **not** be re-rendered because it is a common UI between the two routes.

This performance optimization allows navigation between pages that share a layout to be quicker as only the data fetching and rendering for the page has to run, instead of the entire route that could include shared layouts that fetch their own data.

Because `dashboard/layout.tsx` doesn't re-render, the `searchParams` prop in the layout Server Component might become **stale** after navigation.

- Instead, use the Page `searchParams` prop or the `useSearchParams` hook in a Client Component, which is re-rendered on the client with the latest `searchParams`.

Root Layouts

- The `app` directory **must** include a root `app/layout.js`.
- The root layout **must** define `<html>` and `<body>` tags.
 - You should **not** manually add `<head>` tags such as `<title>` and `<meta>` to root layouts. Instead, you should use the [Metadata API](#) which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.
- You can use [route groups](#) to create multiple root layouts.
 - Navigating **across multiple root layouts** will cause a **full page load** (as opposed to a client-side navigation). For example, navigating from `/cart` that uses `app/(shop)/layout.js` to `/blog` that uses `app/(marketing)/layout.js` will cause a full page load. This **only** applies to multiple root layouts.

> Menu

App Router > ... > File Conventions > loading.js

loading.js

A **loading** file can create instant loading states built on [Suspense](#).

By default, this file is a [Server Component](#) - but can also be used as a Client Component through the `"use client"` directive.

A screenshot of a code editor showing a file named "loading.tsx". The code is a simple function that returns a loading message.

```
1 export default function Loading() {  
2   // Or a custom loading skeleton component  
3   return 'Loading...';  
4 }
```

▼



Loading UI components do not accept any parameters.

Good to know

- While designing loading UI, you may find it helpful to use the [React Developer Tools](#) to manually toggle Suspense boundaries.

> Menu

App Router > ... > File Conventions > Metadata Files

Metadata Files API Reference

This section of the docs covers **Metadata file conventions**. File-based metadata can be defined by adding special metadata files to route segments.

Each file convention can be defined using a static file (e.g. `opengraph-image.jpg`), or a dynamic variant that uses code to generate the file (e.g. `opengraph-image.js`).

Once a file is defined, Next.js will automatically serve the file (with hashes in production for caching) and update the relevant head elements with the correct metadata, such as the asset's URL, file type, and image size.

[favicon, apple-icon, and icon](#)

API Reference for the Favicon, Icon and Apple Icon file conventions.

[opengraph-image and twitter-image](#)

API Reference for the Open Graph Image and Twitter Image file conventions.

[robots.txt](#)

API Reference for robots.txt file.

[sitemap.xml](#)

API Reference for the sitemap.xml file.

> Menu

App Router > ... > Metadata Files > favicon, apple-icon, and icon

favicon, apple-icon, and icon

The `favicon`, `apple-icon` or, `icon` file conventions allow you to set icons for your application.

They are useful for adding app icons that appear in places like web browser tabs, phone home screens, and search engine results.

There are two ways to set app icons:

- [Using image files \(.ico, .jpg, .png\)](#)
- [Using code to generate an icon \(.js, .ts, .tsx\)](#)

Image files (.ico, .jpg, .png)

Use an image file to set an app icon by placing a `favicon`, `icon`, or `apple-icon` image file the root `/app` segment.

Next.js will evaluate the file and automatically add the appropriate tags to your app's `<head>` element.

File convention	Supported file types
<code>favicon</code>	<code>.ico</code>
<code>icon</code>	<code>.ico</code> , <code>.jpg</code> , <code>.jpeg</code> , <code>.png</code> , <code>.svg</code>
<code>apple-icon</code>	<code>.jpg</code> , <code>.jpeg</code> , <code>.png</code> ,

favicon

Add a `favicon.ico` image file to the root `/app` route segment.

```
<link rel="icon" href="/favicon.ico" sizes="any" />
```

icon

Add an `icon.(ico|jpg|jpeg|png|svg)` image file to the root `/app` route segment.

 <head> output 

```
1 <link
2   rel="icon"
3   href="/icon?<generated>"
4   type="image/<generated>"
5   sizes="<generated>"
6 />
```

apple-icon

Add an `apple-icon.(jpg|jpeg|png)` image file to the root `/app` route segment.

 <head> output 

```
1 <link
2   rel="apple-touch-icon"
3   href="/apple-icon?<generated>"
4   type="image/<generated>"
5   sizes="<generated>"
6 />
```

Good to know

- You can set multiple icons by adding a number suffix to the file name. For example, `icon1.png`, `icon2.png`, etc. Numbered files will sort lexically.
- App icons can only be set in the root `/app` segment.
- The appropriate `<link>` tags and attributes such as `rel`, `href`, `type`, and `sizes` are determined by the icon type and metadata of the evaluated file.
 - For example, a 32 by 32px `.png` file will have `type="image/png"` and `sizes="32x32"` attributes.
- `sizes="any"` is added to `favicon.ico` output to [avoid a browser bug ↗](#) where an `.ico` icon is favored over `.svg`.

Generate icons using code (.js, .ts, .tsx)

In addition to using [literal image files](#), you can programmatically **generate** icons using code.

Generate an app icon by creating an `icon` or `apple-icon` route that default exports a function.

File convention

Supported file types

`icon`

`.js`, `.ts`, `.tsx`

`apple-icon`

`.js`, `.ts`, `.tsx`

The easiest way to generate an icon is to use the [ImageResponse](#) API from `next/server`.

app/icon.tsx



```
1 import { ImageResponse } from 'next/server';
2
3 // Route segment config
4 export const runtime = 'edge';
5
6 // Image metadata
7 export const size = {
8   width: 32,
9   height: 32,
10 };
11 export const contentType = 'image/png';
12
13 // Image generation
14 export default function Icon() {
15   return new ImageResponse(
16     (
17       // ImageResponse JSX element
18       <div
19         style={{
20           fontSize: 24,
21           background: 'black',
22           width: '100%',
23           height: '100%',
24           display: 'flex',
25           alignItems: 'center',
26           justifyContent: 'center',
27           color: 'white',
28         }}
29       >
30         A
31       </div>
32     ),
33   // ImageResponse options
34   {
35     // For convenience, we can re-use the exported icons size metadata
36     // config to also set the ImageResponse's width and height.
37     ...size,
38   },
39 }
```

```
39   );
40 }
```

📄 <head> output



```
<link rel="icon" href="/icon?<generated>" type="image/png" sizes="32x32" />
```

Good to know

- By default, generated icons are **statically optimized** (generated at build time and cached) unless they use **dynamic functions** or **dynamic data fetching**.
- You can generate multiple icons in the same file using `generateImageMetadata`.
- You cannot generate a `favicon` icon. Use `icon` or a `favicon.ico` file instead.

Props

The default export function receives the following props:

params (optional)

An object containing the **dynamic route parameters** object from the root segment down to the segment `icon` or `apple-icon` is colocated in.

ts app/shop/[slug]/icon.tsx



```
1 export default function Icon({ params }: { params: { slug: string } }) {
2   // ...
3 }
```

Route

URL

params

app/shop/icon.js

/shop

undefined

app/shop/[slug]/icon.js

/shop/1

{ slug: '1' }

app/shop/[tag]/[item]/icon.js

/shop/1/2

{ tag: '1', item: '2' }

app/shop/...slug/icon.js

/shop/1/2

{ slug: ['1', '2'] }

Returns

The default export function should return a `Blob` | `ArrayBuffer` | `TypedArray` | `DataView` | `ReadableStream` | `Response`.

Note: `ImageResponse` satisfies this return type.

Config exports

You can optionally configure the icon's metadata by exporting `size` and `contentType` variables from the `icon` or `apple-icon` route.

Option	Type
<code>size</code>	<code>{ width: number; height: number }</code>
<code>contentType</code>	<code>string</code> - image MIME type ↗

size

TS icon.tsx / apple-icon.tsx

```
1 export const size = { width: 32, height: 32 };
2
3 export default function Icon() {}
```

📄 <head> output

```
<link rel="icon" sizes="32x32" />
```

contentType

TS icon.tsx / apple-icon.tsx

```
1 export const contentType = 'image/png';
2
3 export default function Icon() {}
```

📄 <head> output

```
<link rel="icon" type="image/png" />
```

Route Segment Config

`icon` and `apple-icon` are specialized [Route Handlers](#) that can use the same [route segment configuration](#) options as Pages and Layouts.

Option	Type	Default
dynamic	'auto' 'force-dynamic' 'error' 'force-static'	'auto'
revalidate	false 'force-cache' 0 number	false
runtime	'nodejs' 'edge'	'nodejs'
preferredRegion	'auto' 'global' 'home' string string[]	'auto'

TS app/icon.tsx

```
1 export const runtime = 'edge';
2
3 export default function Icon() {}
```

> Menu

App Router > ... > Metadata Files > opengraph-image and twitter-image

opengraph-image and twitter-image

The `opengraph-image` and `twitter-image` file conventions allow you to set Open Graph and Twitter images for a route segment.

They are useful for setting the images that appear on social networks and messaging apps when a user shares a link to your site.

There are two ways to set Open Graph and Twitter images:

- [Using image files \(.jpg, .png, .gif\)](#)
- [Using code to generate images \(.js, .ts, .tsx\)](#)

Image files (.jpg, .png, .gif)

Use an image file to set a route segment's shared image by placing an `opengraph-image` or `twitter-image` image file in the segment.

Next.js will evaluate the file and automatically add the appropriate tags to your app's `<head>` element.

File convention	Supported file types
<code>opengraph-image</code>	<code>.jpg</code> , <code>.jpeg</code> , <code>.png</code> , <code>.gif</code>
<code>twitter-image</code>	<code>.jpg</code> , <code>.jpeg</code> , <code>.png</code> , <code>.gif</code>
<code>opengraph-image.alt</code>	<code>.txt</code>
<code>twitter-image.alt</code>	<code>.txt</code>

opengraph-image

Add an `opengraph-image.(jpg|jpeg|png|gif)` image file to any route segment.

 <head> output 

```
1 <meta property="og:image" content="<generated>" />
2 <meta property="og:image:type" content="<generated>" />
3 <meta property="og:image:width" content="<generated>" />
4 <meta property="og:image:height" content="<generated>" />
```

twitter-image

Add a `twitter-image.(jpg|jpeg|png|gif)` image file to any route segment.

 <head> output 

```
1 <meta name="twitter:image" content="<generated>" />
2 <meta name="twitter:image:type" content="<generated>" />
3 <meta name="twitter:image:width" content="<generated>" />
4 <meta name="twitter:image:height" content="<generated>" />
```

opengraph-image.alt.txt

Add an accompanying `opengraph-image.alt.txt` file in the same route segment as the `opengraph-image.(jpg|jpeg|png|gif)` image it's alt text.

 opengraph-image.alt.txt 

About Acme

 <head> output 

```
<meta property="og:image:alt" content="About Acme" />
```

twitter-image.alt.txt

Add an accompanying `twitter-image.alt.txt` file in the same route segment as the `twitter-image.(jpg|jpeg|png|gif)` image it's alt text.

 twitter-image.alt.txt 

About Acme

```
<meta property="og:image:alt" content="About Acme" />
```

Generate images using code (.js, .ts, .tsx)

In addition to using [literal image files](#), you can programmatically **generate** images using code.

Generate a route segment's shared image by creating an [opengraph-image](#) or [twitter-image](#) route that default exports a function.

File convention	Supported file types
opengraph-image	.js, .ts, .tsx
twitter-image	.js, .ts, .tsx

Good to know

- By default, generated images are [statically optimized](#) (generated at build time and cached) unless they use [dynamic functions](#) or [dynamic data fetching](#).
- You can generate multiple Images in the same file using [generateImageMetadata](#).

The easiest way to generate an image is to use the [ImageResponse](#) API from [next/server](#).

```
1 import { ImageResponse } from 'next/server';
2
3 // Route segment config
4 export const runtime = 'edge';
5
6 // Image metadata
7 export const alt = 'About Acme';
8 export const size = {
9   width: 1200,
10  height: 630,
11 };
12
13 export const contentType = 'image/png';
14
15 // Font
16 const interSemiBold = fetch(
```

```

17   new URL('./Inter-SemiBold.ttf', import.meta.url),
18 ).then((res) => res.arrayBuffer());
19
20 // Image generation
21 export default function Image() {
22   return new ImageResponse(
23   (
24     // ImageResponse JSX element
25     <div
26       style={{
27         fontSize: 128,
28         background: 'white',
29         width: '100%',
30         height: '100%',
31         display: 'flex',
32         alignItems: 'center',
33         justifyContent: 'center',
34       }}
35     >
36       About Acme
37     </div>
38   ),
39   // ImageResponse options
40   {
41     // For convenience, we can re-use the exported opengraph-image
42     // size config to also set the ImageResponse's width and height.
43     ...size,
44     fonts: [
45       {
46         name: 'Inter',
47         data: await interSemiBold,
48         style: 'normal',
49         weight: 400,
50       },
51     ],
52   },
53 );
54 }

```

 <head> output

```

1 <meta property="og:image" content="<generated>" />
2 <meta property="og:image:alt" content="About Acme" />
3 <meta property="og:image:type" content="image/png" />
4 <meta property="og:image:width" content="1200" />
5 <meta property="og:image:height" content="630" />

```

Props

The default export function receives the following props:

params (optional)

An object containing the [dynamic route parameters](#) object from the root segment down to the segment `opengraph-image` or `twitter-image` is colocated in.

TS app/shop/[slug]/opengraph-image.tsx

```
1 export default function Image({ params }: { params: { slug: string } }) {  
2   // ...  
3 }
```

Route	URL	params
app/shop/opengraph-image.js	/shop	undefined
app/shop/[slug]/opengraph-image.js	/shop/1	{ slug: '1' }
app/shop/[tag]/[item]/opengraph-image.js	/shop/1/2	{ tag: '1', item: '2' }
app/shop/...slug/opengraph-image.js	/shop/1/2	{ slug: ['1', '2'] }

Returns

The default export function should return a `Blob` | `ArrayBuffer` | `TypedArray` | `DataView` | `ReadableStream` | `Response`.

Note: `ImageResponse` satisfies this return type.

Config exports

You can optionally configure the image's metadata by exporting `alt`, `size`, and `contentType` variables from `opengraph-image` or `twitter-image` route.

Option	Type
<code>alt</code>	<code>string</code>
<code>size</code>	<code>{ width: number; height: number }</code>
<code>contentType</code>	<code>string</code> - image MIME type ↗
<code>alt</code>	

TS opengraph-image.tsx / twitter-image.tsx

```
1 export const alt = 'My images alt text';
2
3 export default function Image() {}
```

<head> output

```
<meta property="og:image:alt" content="My images alt text" />
```

size

opengraph-image.tsx / twitter-image.tsx

```
1 export const size = { width: 1200, height: 630 };
2
3 export default function Image() {}
```

<head> output

```
1 <meta property="og:image:width" content="1200" />
2 <meta property="og:image:height" content="630" />
```

contentType

opengraph-image.tsx / twitter-image.tsx

```
1 export const contentType = 'image/png';
2
3 export default function Image() {}
```

<head> output

```
<meta property="og:image:type" content="image/png" />
```

Route Segment Config

`opengraph-image` and `twitter-image` are specialized [Route Handlers](#) that can use the same [route segment configuration](#) options as Pages and Layouts.

Option	Type	Default
<code>dynamic</code>	<code>'auto' 'force-dynamic' 'error' 'force-static'</code>	<code>'auto'</code>

Option	Type	Default
revalidate	false 'force-cache' 0 number	false
runtime	'nodejs' 'edge'	'nodejs'
preferredRegion	'auto' 'global' 'home' string string[]	'auto'

TS app/opengraph-image.tsx

```
1 export const runtime = 'edge';
2
3 export default function Image() {}
```

Examples

Using external data

This example uses the `params` object and external data to generate the image.

Good to know: By default, this generated image will be [statically optimized](#). You can configure the individual `fetch options` or route segments `options` to change this behavior.

TS app/posts/[slug]/opengraph-image.tsx

```
1 import { ImageResponse } from 'next/server';
2
3 export const runtime = 'edge';
4
5 export const alt = 'About Acme';
6 export const size = {
7   width: 1200,
8   height: 630,
9 };
10 export const contentType = 'image/png';
11
12 export default async function Image({ params }: { params: { slug: string } }) {
13   const post = await fetch(`https://.../posts/${params.slug}`).then((res) =>
14     res.json(),
15   );
16
17   return new ImageResponse(
18     (
19       <div
20         style={{
21           fontSize: 48,
22           background: 'white',
23           width: '100%',
```

```
24         height: '100%',
25         display: 'flex',
26         alignItems: 'center',
27         justifyContent: 'center',
28     )}
29     >
30     {post.title}
31     </div>
32 ),
33 {
34     ...size,
35 },
36 );
37 }
```

> Menu

App Router > ... > Metadata Files > robots.txt

robots.txt

Add or generate a `robots.txt` file that matches the [Robots Exclusion Standard](#) ↗ in the **root** of `app` directory to tell search engine crawlers which URLs they can access on your site.

Static `robots.txt`

 app/robots.txt

```
1 User-Agent: *
2 Allow: /
3 Disallow: /private/
4
5 Sitemap: https://acme.com/sitemap.xml
```

Generate a Robots file

Add a `robots.js` or `robots.ts` file that returns a [Robots object](#).

 app/robots.ts

▼



```
1 import { MetadataRoute } from 'next';
2
3 export default function robots(): MetadataRoute.Robots {
4   return {
5     rules: [
6       userAgent: '*',
7       allow: '/',
8       disallow: '/private/',
9     },
10   },
11 }
```

```
10     sitemap: 'https://acme.com/sitemap.xml',
11 };
12 }
```

Output:

```
1 User-Agent: *
2 Allow: /
3 Disallow: /private/
4
5 Sitemap: https://acme.com/sitemap.xml
```

Robots object

```
1 type Robots = {
2   rules:
3   | {
4     userAgent?: string | string[];
5     allow?: string | string[];
6     disallow?: string | string[];
7     crawlDelay?: number;
8   }
9   | Array<{
10     userAgent: string | string[];
11     allow?: string | string[];
12     disallow?: string | string[];
13     crawlDelay?: number;
14   }>;
15   sitemap?: string | string[];
16   host?: string;
17 };
```

> Menu

App Router > ... > Metadata Files > sitemap.xml

sitemap.xml

Add or generate a `sitemap.xml` file that matches the [Sitemaps XML format](#) in the **root** of `app` directory to help search engine crawlers crawl your site more efficiently.

Static `sitemap.xml`

📄 app/sitemap.xml

🔗

```
1 <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
2   <url>
3     <loc>https://acme.com</loc>
4     <lastmod>2023-04-06T15:02:24.021Z</lastmod>
5   </url>
6   <url>
7     <loc>https://acme.com/about</loc>
8     <lastmod>2023-04-06T15:02:24.021Z</lastmod>
9   </url>
10  <url>
11    <loc>https://acme.com/blog</loc>
12    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
13  </url>
14 </urlset>
```

Generate a Sitemap

Add a `sitemap.js` or `sitemap.ts` file that returns `Sitemap`.

typescript app/sitemap.ts

▼

🔗

```
1 import { MetadataRoute } from 'next';
2
3 export default function sitemap(): MetadataRoute.Sitemap {
4   return [
5     {
6       url: 'https://acme.com',
7       lastModified: new Date(),
8     },
9     {
10       url: 'https://acme.com/about',
11       lastModified: new Date(),
12     },
13     {
14       url: 'https://acme.com/blog',
15       lastModified: new Date(),
16     },
17   ];
18 }
```

Output:

acme.com/sitemap.xml

```
1 <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
2   <url>
3     <loc>https://acme.com</loc>
4     <lastmod>2023-04-06T15:02:24.021Z</lastmod>
5   </url>
6   <url>
7     <loc>https://acme.com/about</loc>
8     <lastmod>2023-04-06T15:02:24.021Z</lastmod>
9   </url>
10  <url>
11    <loc>https://acme.com/blog</loc>
12    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
13  </url>
14 </urlset>
```

Sitemap Return Type

```
1 type Sitemap = Array<{
2   url: string;
3   lastModified?: string | Date;
4 }>;
```

Good to know

- In the future, we will support multiple sitemaps and sitemap indexes.

> Menu

App Router > ... > File Conventions > not-found.js

not-found.js

The **not-found** file is used to render UI when the `notFound` function is thrown within a route segment. Along with serving a custom UI, Next.js will also return a `404` HTTP status code.

TS app/blog/not-found.tsx

```
1 import Link from 'next/link';
2
3 export default function NotFound() {
4   return (
5     <div>
6       <h2>Not Found</h2>
7       <p>Could not find requested resource</p>
8       <p>
9         View <Link href="/blog">all posts</Link>
10      </p>
11    </div>
12  );
13}
```

Note: In addition to catching expected `notFound()` errors, the root `app/not-found.js` file also handles any unmatched URLs for your whole application. This means users that visit a URL that is not handled by your app will be shown the UI exported by the `app/not-found.js` file.

Props

`not-found.js` components do not accept any props.

> Menu

App Router > ... > File Conventions > page.js

page.js

A **page** is UI that is unique to a route.

TS app/blog/[slug]/page.tsx

```
1 export default function Page({  
2   params,  
3   searchParams,  
4 }: {  
5   params: { slug: string };  
6   searchParams: { [key: string]: string | string[] | undefined };  
7 }) {  
8   return <h1>My Page</h1>;  
9 }
```

Props

params (optional)

An object containing the [dynamic route parameters](#) from the root segment down to that page. For example:

Example

URL

params

app/shop/[slug]/page.js

/shop/1

{ slug: '1' }

app/shop/[category]/[item]/page.js

/shop/1/2

{ category: '1', item: '2' }

app/shop/...slug/page.js

/shop/1/2

{ slug: ['1', '2'] }

searchParams (optional)

An object containing the [search parameters](#) of the current URL. For example:

URL

searchParams

/shop?a=1

{ a: '1' }

/shop?a=1&b=2

{ a: '1', b: '2' }

/shop?a=1&a=2

{ a: ['1', '2'] }

Good to know:

- `searchParams` is a **Dynamic API** whose values cannot be known ahead of time. Using it will opt the page into **dynamic rendering** at request time.
- `searchParams` returns a plain JavaScript object and not a `URLSearchParams` instance.

> Menu

App Router > ... > File Conventions > route.js

route.js

Route Handlers allow you to create custom request handlers for a given route using the [Web Request ↗](#) and [Response ↗](#) APIs.

HTTP Methods

A **route** file allows you to create custom request handlers for a given route. The following [HTTP methods ↗](#) are supported: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, and `OPTIONS`.

TS route.ts

▼



```
1 export async function GET(request: Request) {}
2
3 export async function HEAD(request: Request) {}
4
5 export async function POST(request: Request) {}
6
7 export async function PUT(request: Request) {}
8
9 export async function DELETE(request: Request) {}
10
11 export async function PATCH(request: Request) {}
12
13 // If `OPTIONS` is not defined, Next.js will automatically implement `OPTIONS` and set t
14 export async function OPTIONS(request: Request) {}
```

Good to know: Route Handlers are only available inside the `app` directory. You **do not** need to use API Routes ([pages](#)) and Route Handlers (`app`) together, as Route Handlers should be able to handle all use cases.

Parameters

request (optional)

The `request` object is a [NextRequest](#) object, which is an extension of the Web [Request](#) API.

`NextRequest` gives you further control over the incoming request, including easily accessing `cookies` and an extended, parsed, URL object `nextUrl`.

context (optional)

`js` app/dashboard/[team]/route.js

```
1 export async function GET(request, context: { params }) {
2   const team = params.team; // '1'
3 }
```



Currently, the only value of `context` is `params`, which is an object containing the [dynamic route parameters](#) for the current route.

Example	URL	params
app/dashboard/[team]/route.js	/dashboard/1	{ team: '1' }
app/shop/[tag]/[item]/route.js	/shop/1/2	{ tag: '1', item: '2' }
app/blog/...slug/route.js	/blog/1/2	{ slug: ['1', '2'] }

NextResponse

Route Handlers can extend the Web Response API by returning a `NextResponse` object. This allows you to easily set cookies, headers, redirect, and rewrite. [View the API reference](#).

> Menu

App Router > ... > File Conventions > Route Segment Config

Route Segment Config

The Route Segment options allows you configure the behavior of a [Page](#), [Layout](#), or [Route Handler](#) by directly exporting the following variables:

Option	Type	Default
<code>dynamic</code>	<code>'auto' 'force-dynamic' 'error' 'force-static'</code>	<code>'auto'</code>
<code>dynamicParams</code>	<code>boolean</code>	<code>true</code>
<code>revalidate</code>	<code>false 'force-cache' 0 number</code>	<code>false</code>
<code>fetchCache</code>	<code>'auto' 'default-cache' 'only-cache' 'force-cache' 'force-no-store' 'default-no-store' 'only-no-store'</code>	<code>'auto'</code>
<code>runtime</code>	<code>'nodejs' 'edge'</code>	<code>'nodejs'</code>
<code>preferredRegion</code>	<code>'auto' 'global' 'home' string string[]</code>	<code>'auto'</code>

TS layout.tsx / page.tsx / route.ts

```
1 export const dynamic = 'auto';
2 export const dynamicParams = true;
3 export const revalidate = false;
4 export const fetchCache = 'auto';
5 export const runtime = 'nodejs';
6 export const preferredRegion = 'auto';
7
8 export default function MyComponent() {}
```

Good to know:

- The values of the config options currently need be statically analyzable. For example `revalidate = 600` is valid, but `revalidate = 60 * 10` is not.

Options

dynamic

Change the dynamic behavior of a layout or page to fully static or fully dynamic.

layout.tsx / page.tsx / route.ts

```
1 export const dynamic = 'auto';
2 // 'auto' | 'force-dynamic' | 'error' | 'force-static'
```

Migration Note: The new model in the `app` directory favors granular caching control at the `fetch` request level over the binary all-or-nothing model of `getServerSideProps` and `getStaticProps` at the page-level in the `pages` directory. The `dynamic` option is a way to opt back in to the previous model as a convenience and provides a simpler migration path.

- **'auto'** (default): The default option to cache as much as possible without preventing any components from opting into dynamic behavior.
- **'force-dynamic'**: Force dynamic rendering and dynamic data fetching of a layout or page by disabling all caching of `fetch` requests and always revalidating. This option is equivalent to:
 - `getServerSideProps()` in the `pages` directory.
 - Setting the option of every `fetch()` request in a layout or page to `{ cache: 'no-store', next: { revalidate: 0 } }`.
 - Setting the segment config to `export const fetchCache = 'force-no-store'`
- **'error'**: Force static rendering and static data fetching of a layout or page by causing an error if any components use [dynamic functions](#) or [dynamic fetches](#). This option is equivalent to:
 - `getStaticProps()` in the `pages` directory.
 - Setting the option of every `fetch()` request in a layout or page to `{ cache: 'force-cache' }`.
 - Setting the segment config to `fetchCache = 'only-cache', dynamicParams = false`.
 - Note: `dynamic = 'error'` changes the default of `dynamicParams` from `true` to `false`. You can opt back into dynamically rendering pages for dynamic params not generated by `generateStaticParams` by manually setting `dynamicParams = true`.
- **'force-static'**: Force static rendering and static data fetching of a layout or page by forcing `cookies()`, `headers()` and `useSearchParams()` to return empty values.

Good to know:

- Instructions on [how to migrate](#) from `getServerSideProps` and `getStaticProps` to `dynamic: 'force-dynamic'` and `dynamic: 'error'` can be found in the [upgrade guide](#).

dynamicParams

Control what happens when a dynamic segment is visited that was not generated with [generateStaticParams](#).

TS layout.tsx / page.tsx

```
export const dynamicParams = true; // true | false,
```

- `true` (default): Dynamic segments not included in `generateStaticParams` are generated on demand.
- `false`: Dynamic segments not included in `generateStaticParams` will return a 404.

Good to know:

- This option replaces the `fallback: true | false | blocking` option of `getStaticPaths` in the `pages` directory.
- When `dynamicParams = true`, the segment uses [Streaming Server Rendering](#).
- If the `dynamic = 'error'` and `dynamic = 'force-static'` are used, it'll change the default of `dynamicParams` to `false`.

revalidate

Set the default revalidation time for a layout or page. This option does not override the `revalidate` value set by individual `fetch` requests.

TS layout.tsx / page.tsx / route.ts

```
1 export const revalidate = false;
2 // false | 'force-cache' | 0 | number
```

- `false`: (default) The default heuristic to cache any `fetch` requests that set their `cache` option to `'force-cache'` or are discovered before a [dynamic function](#) is used. Semantically equivalent to `revalidate: Infinity` which effectively means the resource should be cached indefinitely. It is still possible for individual `fetch` requests to use `cache: 'no-store'` or `revalidate: 0` to avoid being cached and make the route dynamically rendered. Or set `revalidate` to a positive number lower than the route default to increase the revalidation frequency of a route.

- **0** : Ensure a layout or page is always [dynamically rendered](#) even if no dynamic functions or dynamic data fetches are discovered. This option changes the default of `fetch` requests that do not set a `cache` option to `'no-store'` but leaves `fetch` requests that opt into `'force-cache'` or use a positive `revalidate` as is.
- **number** : (in seconds) Set the default revalidation frequency of a layout or page to `n` seconds.

Revalidation Frequency

- The lowest `revalidate` across each layout and page of a single route will determine the revalidation frequency of the *entire* route. This ensures that child pages are revalidated as frequently as their parent layouts.
- Individual `fetch` requests can set a lower `revalidate` than the route's default `revalidate` to increase the revalidation frequency of the entire route. This allows you to dynamically opt-in to more frequent revalidation for certain routes based on some criteria.

fetchCache

► **This is an advanced option that should only be used if you specifically need to override the default behavior.**

runtime

```
TS layout.tsx / page.tsx / route.ts
```

```
1 export const runtime = 'nodejs';
2 // 'edge' | 'nodejs'
```

- `nodejs` (default)
- `edge`

Learn more about the [Edge and Node.js runtimes](#).

preferredRegion

```
TS layout.tsx / page.tsx / route.ts
```

```
1 export const preferredRegion = 'auto';
2 // 'auto' | 'global' | 'home' | ['iad1', 'sfo1']
```

Support for `preferredRegion`, and regions supported, is dependent on your deployment platform.

Good to know:

- If a `preferredRegion` is not specified, it will inherit the option of the nearest parent layout.
- The root layout defaults to `all` regions.

generateStaticParams

The `generateStaticParams` function can be used in combination with [dynamic route segments](#) to define the list of route segment parameters that will be statically generated at build time instead of on-demand at request time.

See the [API reference](#) for more details.

> Menu

App Router > ... > File Conventions > template.js

template.js

This documentation is still being written. Please check back later.

> Menu

App Router > API Reference > Functions

Functions

[cookies](#)

API Reference for the cookies function.

[draftMode](#)

API Reference for the draftMode function.

[fetch](#)

API reference for the extended fetch function.

[generateImageMetadata](#)

Learn how to generate multiple images in a single Metadata API special file.

[generateMetadata](#)

Learn how to add Metadata to your Next.js application for improved search engine optimization (SEO) and web shareability.

[generateStaticParams](#)

API reference for the generateStaticParams function.

headers

API reference for the headers function.

ImageResponse

API Reference for the ImageResponse constructor.

NextRequest

API Reference for NextRequest.

NextResponse

API Reference for NextResponse.

notFound

API Reference for the notFound function.

redirect

API Reference for the redirect function.

revalidatePath

API Reference for the revalidatePath function.

revalidateTag

API Reference for the revalidateTag function.

useParams

[usePathname](#)

API Reference for the usePathname hook.

[useReportWebVitals](#)

API Reference for the useReportWebVitals function.

[useRouter](#)

API reference for the useRouter hook.

[useSearchParams](#)

API Reference for the useSearchParams hook.

[useSelectedLayoutSegment](#)

API Reference for the useSelectedLayoutSegment hook.

[useSelectedLayoutSegments](#)

API Reference for the useSelectedLayoutSegments hook.

> Menu

App Router > ... > Functions > cookies

cookies

The `cookies` function allows you to read the HTTP incoming request cookies from a [Server Component](#) or write outgoing request cookies in a [Server Action](#) or [Route Handler](#).

Good to know:

- `cookies()` is a [Dynamic Function](#) whose returned values cannot be known ahead of time. Using it in a layout or page will opt a route into [dynamic rendering](#) at request time.

cookies().get(name)

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.

JS app/page.js



```
1 import { cookies } from 'next/headers';
2
3 export default function Page() {
4   const cookieStore = cookies();
5   const theme = cookieStore.get('theme');
6   return '...';
7 }
```

cookies().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.

```
1 import { cookies } from 'next/headers';
2
3 export default function Page() {
4   const cookieStore = cookies();
5   return cookieStore.getAll().map((cookie) => (
6     <div key={cookie.name}>
7       <p>Name: {cookie.name}</p>
8       <p>Value: {cookie.value}</p>
9     </div>
10   )));
11 }
```

cookies().has(name)

A method that takes a cookie name and returns a `boolean` based on if the cookie exists (`true`) or not (`false`).

```
1 import { cookies } from 'next/headers';
2
3 export default function Page() {
4   const cookiesList = cookies();
5   const hasCookie = cookiesList.has('theme');
6   return '...';
7 }
```

cookies().set(name, value, options)

A method that takes a cookie name, value, and options and sets the outgoing request cookie.

Good to know: `.set()` is only available in a [Server Action](#) or [Route Handler](#).

```
1 'use server';
```

```
2
3 import { cookies } from 'next/headers';
4
5 async function create(data) {
6   cookies().set('name', 'lee');
7   // or
8   cookies().set('name', 'lee', { secure: true });
9   // or
10  cookies().set({
11    name: 'name',
12    value: 'lee',
13    httpOnly: true,
14    path: '/',
15  });
16 }
```

Deleting cookies

To "delete" a cookie, you must set a new cookie with the same name and an empty value. You can also set the `maxAge` to `0` to expire the cookie immediately.

`.set()` is only available in a [Server Action](#) or [Route Handler](#).

JS app/actions.js

```
1 'use server';
2
3 import { cookies } from 'next/headers';
4
5 async function create(data) {
6   cookies().set({
7     name: 'name',
8     value: '',
9     expires: new Date('2016-10-05')
10    path: '/', // For all paths
11  });
12 }
```

You can only set cookies that belong to the same domain from which `.set()` is called. Additionally, the code must be executed on the same protocol (HTTP or HTTPS) as the cookie you want to update.

Next Steps

For more information on what to do next, we recommend the following sections

App Router > ... > Data Fetching

Server Actions

Use Server Actions to mutate data in your Next.js application.

> Menu

App Router > ... > Functions > draftMode

draftMode

The `draftMode` function allows you to detect [Draft Mode](#) inside a [Server Component](#).

JS app/page.js



```
1 import { draftMode } from 'next/headers';
2
3 export default function Page() {
4   const { isEnabled } = draftMode();
5   return (
6     <main>
7       <h1>My Blog Post</h1>
8       <p>Draft Mode is currently {isEnabled ? 'Enabled' : 'Disabled'}</p>
9     </main>
10   );
11 }
```


> Menu

App Router > ... > Functions > fetch

fetch

Next.js extends the native [Web `fetch\(\)` API](#) to allow each request on the server to set its own persistent caching semantics.

In the browser, the `cache` option indicates how a `fetch` request will interact with the *browser's* HTTP cache. With this extension, `cache` indicates how a *server-side* `fetch` request will interact with the framework's persistent HTTP cache.

You can call `fetch` with `async` and `await` directly within 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 }
```

fetch(url, options)

Since Next.js extends the [Web fetch\(\) API](#), you can use any of the [native options available](#).

Further, Next.js polyfills `fetch` on both the client and the server, so you can use `fetch` in both [Server and Client Components](#).

options.cache

Configure how the request should interact with Next.js HTTP cache.

```
fetch(`https://...`, { cache: 'force-cache' | 'no-store' });
```

- **force-cache** (default) - Next.js looks for a matching request in its HTTP cache.
 - If there is a match and it is fresh, it will be returned from the cache.
 - If there is no match or a stale match, Next.js will fetch the resource from the remote server and update the cache with the downloaded resource.
- **no-store** - Next.js fetches the resource from the remote server on every request without looking in the cache, and it will not update the cache with the downloaded resource.

Good to know:

- If you don't provide a `cache` option, Next.js will default to `force-cache`, unless a [dynamic function](#) such as `cookies()` is used, in which case it will default to `no-store`.
- The `no-cache` option behaves the same way as `no-store` in Next.js.

options.next.revalidate

```
fetch(`https://...`, { next: { revalidate: false | 0 | number } });
```

Set the cache lifetime of a resource (in seconds).

- **false** - Cache the resource indefinitely. Semantically equivalent to `revalidate: Infinity`. The [HTTP cache](#) may evict older resources over time.
- **0** - Prevent the resource from being cached.
- **number** - (in seconds) Specify the resource should have a cache lifetime of at most `n` seconds.

Good to know:

- If an individual `fetch()` request sets a `revalidate` number lower than the `default revalidate` of a route, the whole route revalidation interval will be decreased.

- If two fetch requests with the same URL in the same route have different `revalidate` values, the lower value will be used.
- As a convenience, it is not necessary to set the `cache` option if `revalidate` is set to a number since `0` implies `cache: 'no-store'` and a positive value implies `cache: 'force-cache'`.
- Conflicting options such as `{ revalidate: 0, cache: 'force-cache' }` or `{ revalidate: 10, cache: 'no-store' }` will cause an error.

> Menu

App Router > ... > Functions > generateImageMetadata

generateImageMetadata

You can use `generateImageMetadata` to generate different versions of one image or return multiple images for one route segment. This is useful for when you want to avoid hard-coding metadata values, such as for icons.

Parameters

`generateMetadata` function accepts the following parameters:

params (optional)

An object containing the [dynamic route parameters](#) object from the root segment down to the segment `generateImageMetadata` is called from.

TS icon.tsx

```
1 export function generateImageMetadata({  
2   params,  
3 }: {  
4   params: { slug: string };  
5 }) {  
6   // ...  
7 }
```

Route	URL	params
app/shop/icon.js	/shop	undefined
app/shop/[slug]/icon.js	/shop/1	{ slug: '1' }
app/shop/[tag]/[item]/icon.js	/shop/1/2	{ tag: '1', item: '2' }
app/shop/[^.]+/icon.js	/shop/1/2	{ slug: ['1', '2'] }

Returns

The `generateImageMetadata` function should return an `array` of objects containing the image's metadata such as `alt` and `size`. In addition, each item **must** include an `id` value will be passed to the props of the image generating function.

Image Metadata Object

Type

<code>id</code>	<code>string</code> (required)
-----------------	--------------------------------

<code>alt</code>	<code>string</code>
------------------	---------------------

<code>size</code>	<code>{ width: number; height: number }</code>
-------------------	--

<code>contentType</code>	<code>string</code>
--------------------------	---------------------

TS icon.tsx

▼



```
1 import { ImageResponse } from 'next/server';
2
3 export function generateImageMetadata() {
4   return [
5     {
6       contentType: 'image/png',
7       size: { width: 48, height: 48 },
8       id: 'small',
9     },
10    {
11      contentType: 'image/png',
12      size: { width: 72, height: 72 },
13      id: 'medium',
14    },
15  ];
16}
17
18 export default function Icon({ id }: { id: string }) {
19   return new ImageResponse(
20     (
21       <div
22         style={{
23           width: '100%',
24           height: '100%',
25           display: 'flex',
26           alignItems: 'center',
27           justifyContent: 'center',
28           fontSize: 88,
29           background: '#000',
30           color: '#fafafa',
31         }})
```

```
32      >
33      Icon {id}
34    </div>
35  ),
36 );
37 }
```

Examples

Using external data

This example uses the `params` object and external data to generate multiple [Open Graph images](#) for a route segment.

ts app/products/[id]/opengraph-image.tsx



```
1 import { ImageResponse } from 'next/server';
2 import { getCaptionForImage, getOGImages } from '@/app/utils/images';
3
4 export async function generateImageMetadata({
5   params,
6 }: {
7   params: { id: string };
8 }) {
9   const images = await getOGImages(params.id);
10
11   return images.map((image, idx) => ({
12     id: idx,
13     size: { width: 1200, height: 600 },
14     alt: image.text,
15     contentType: 'image/png',
16   }));
17 }
18
19 export default async function Image({
20   params,
21   id,
22 }: {
23   params: { id: string };
24   id: number;
25 }) {
26   const productId = params.id;
27   const imageId = id;
28   const text = await getCaptionForImage(productId, imageId);
29
30   return new ImageResponse(
31     (
32       <div
33         style={
34           {
35             // ...
36           }
37     }
38   )
39   )
40 }
```

```
37      }
38      >
39      {text}
40      </div>
41    ),
42  );
43 }
```

Next Steps

View all the Metadata API options.

App Router > ... > File Conventions

Metadata Files

API documentation for the metadata file conventions.

App Router > ... > Optimizing

Metadata

Use the Metadata API to define metadata in any layout or page.

> Menu

App Router > ... > Functions > generateMetadata

Metadata Object and generateMetadata Options

This page covers all **Config-based Metadata** options with `generateMetadata` and the static metadata object.

TS layout.tsx / page.tsx

```
1 import { Metadata } from 'next';
2
3 // either Static metadata
4 export const metadata: Metadata = {
5   title: '...',
6 };
7
8 // or Dynamic metadata
9 export async function generateMetadata({ params }) {
10   return {
11     title: '...',
12   };
13 }
```

Good to know:

- The `metadata` object and `generateMetadata` function exports are **only supported in Server Components**.
- You cannot export both the `metadata` object and `generateMetadata` function from the same route segment.

The `metadata` object

To define static metadata, export a `Metadata` object from a `layout.js` or `page.js` file.

TS layout.tsx / page.tsx

```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: '...',
5   description: '...',
6 };
7
8 export default function Page() {}
```

See the [Metadata Fields](#) for a complete list of supported options.

generateMetadata function

Dynamic metadata depends on **dynamic information**, such as the current route parameters, external data, or `metadata` in parent segments, can be set by exporting a `generateMetadata` function that returns a `Metadata` object.

TS app/products/[id]/page.tsx

⌄



```
1 import { Metadata, ResolvingMetadata } from 'next';
2
3 type Props = {
4   params: { id: string };
5   searchParams: { [key: string]: string | string[] | undefined };
6 };
7
8 export async function generateMetadata(
9   { params, searchParams }: Props,
10   parent: ResolvingMetadata,
11 ): Promise<Metadata> {
12   // read route params
13   const id = params.id;
14
15   // fetch data
16   const product = await fetch(`https://.../${id}`).then((res) => res.json());
17
18   // optionally access and extend (rather than replace) parent metadata
19   const previousImages = (await parent).openGraph?.images || [];
20
21   return {
22     title: product.title,
23     openGraph: {
24       images: ['/some-specific-page-image.jpg', ...previousImages],
25     },
26   };
27 }
28
```

```
29 export default function Page({ params, searchParams }: Props) {}
```

Parameters

`generateMetadata` function accepts the following parameters:

- `props` - An object containing the parameters of the current route:
 - `params` - An object containing the [dynamic route parameters](#) object from the root segment down to the segment `generateMetadata` is called from. Examples:

Route	URL	params
app/shop/[slug]/page.js	/shop/1	{ slug: '1' }
app/shop/[tag]/[item]/page.js	/shop/1/2	{ tag: '1', item: '2' }
app/shop/[...slug]/page.js	/shop/1/2	{ slug: ['1', '2'] }

- `searchParams` - An object containing the current URL's [search params ↗](#). Examples:

URL	searchParams
/shop?a=1	{ a: '1' }
/shop?a=1&b=2	{ a: '1', b: '2' }
/shop?a=1&a=2	{ a: ['1', '2'] }

- `parent` - A promise of the resolved metadata from parent route segments.

Returns

`generateMetadata` should return a [Metadata object](#) containing one or more metadata fields.

Good to know:

- If metadata doesn't depend on runtime information, it should be defined using the static [metadata object](#) rather than `generateMetadata`.
- When rendering a route, Next.js will [automatically deduplicate fetch requests](#) for the same data across `generateMetadata`, `generateStaticParams`, Layouts, Pages, and Server Components. React [cache](#) can be used if `fetch` is unavailable.
- `searchParams` are only available in `page.js` segments.
- The [redirect\(\)](#) and [notFound\(\)](#) Next.js methods can also be used inside `generateMetadata`.

Metadata Fields

title

The `title` attribute is used to set the title of the document. It can be defined as a simple [string](#) or an optional [template object](#).

String

`js` layout.js / page.js

```
1 export const metadata = {  
2   title: 'Next.js',  
3 };
```

`□` <head> output

```
<title>Next.js</title>
```

Template object

`ts` app/layout.tsx

```
1 import { Metadata } from 'next';  
2  
3 export const metadata: Metadata = {  
4   title: {  
5     template: '...',  
6     default: '...',  
7     absolute: '...',  
8   },  
9 };
```

Default

`title.default` can be used to provide a **fallback title** to child route segments that don't define a `title`.

`ts` app/layout.tsx

```
1 export const metadata = {  
2   title: {  
3     default: 'Acme',  
4   },  
5 }
```

```
5 };
```

TS app/about/page.tsx

```
1 export const metadata = {};
2
3 // Output: <title>Acme</title>
```

Template

`title.template` can be used to add a prefix or a suffix to `titles` defined in **child** route segments.

TS app/layout.tsx

```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: {
5     template: '%s | Acme',
6     default: 'Acme', // a default is required when creating a template
7   },
8 };
```

TS app/about/page.tsx

```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: 'About',
5 };
6
7 // Output: <title>About | Acme</title>
```

Good to know:

- `title.template` applies to **child** route segments and **not** the segment it's defined in. This means:
 - `title.default` is **required** when you add a `title.template`.
 - `title.template` defined in `layout.js` will not apply to a `title` defined in a `page.js` of the same route segment.
 - `title.template` defined in `page.js` has no effect because a page is always the terminating segment (it doesn't have any children route segments).
 - `title.template` has **no effect** if a route has not defined a `title` or `title.default`.

Absolute

`title.absolute` can be used to provide a title that **ignores** `title.template` set in parent segments.

TS app/layout.tsx

```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: {
5     template: '%s | Acme',
6   },
7};
```

TS app/about/page.tsx

```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: {
5     absolute: 'About',
6   },
7};
8
9 // Output: <title>About</title>
```

Good to know:

- `layout.js`
 - `title` (string) and `title.default` define the default title for child segments (that do not define their own `title`). It will augment `title.template` from the closest parent segment if it exists.
 - `title.absolute` defines the default title for child segments. It ignores `title.template` from parent segments.
 - `title.template` defines a new title template for child segments.
- `page.js`
 - If a page does not define its own title the closest parents resolved title will be used.
 - `title` (string) defines the routes title. It will augment `title.template` from the closest parent segment if it exists.
 - `title.absolute` defines the route title. It ignores `title.template` from parent segments.
 - `title.template` has no effect in `page.js` because a page is always the terminating segment of a route.

description

JS layout.js / page.js

```
1 export const metadata = {
```

```
2   description: 'The React Framework for the Web',
3 }
```

📄 <head> output



```
<meta name="description" content="The React Framework for the Web" />
```

Basic Fields

JSImport / page.js



```
1 export const metadata = {
2   generator: 'Next.js',
3   applicationName: 'Next.js',
4   referrer: 'origin-when-cross-origin',
5   keywords: ['Next.js', 'React', 'JavaScript'],
6   authors: [{ name: 'Seb' }, { name: 'Josh', url: 'https://nextjs.org' }],
7   colorScheme: 'dark',
8   creator: 'Jiachi Liu',
9   publisher: 'Sebastian Markbåge',
10  formatDetection: {
11    email: false,
12    address: false,
13    telephone: false,
14  },
15};
```

📄 <head> output



```
1 <meta name="application-name" content="Next.js" />
2 <meta name="author" content="Seb" />
3 <link rel="author" href="https://nextjs.org" />
4 <meta name="author" content="Josh" />
5 <meta name="generator" content="Next.js" />
6 <meta name="keywords" content="Next.js,React,JavaScript" />
7 <meta name="referrer" content="origin-when-cross-origin" />
8 <meta name="color-scheme" content="dark" />
9 <meta name="creator" content="Jiachi Liu" />
10 <meta name="publisher" content="Sebastian Markbåge" />
11 <meta name="format-detection" content="telephone=no, address=no, email=no" />
```

metadataBase

`metadataBase` is a convenience option to set a base URL prefix for `metadata` fields that require a fully qualified URL.

- `metadataBase` allows URL-based `metadata` fields defined in the **current route segment and below** to use a **relative path** instead of an otherwise required absolute URL.
- The field's relative path will be composed with `metadataBase` to form a fully qualified URL.
- If not configured, `metadataBase` is **automatically populated** with a **default value**.

JS layout.js / page.js

```

1  export const metadata = {
2    metadataBase: new URL('https://acme.com'),
3    alternates: {
4      canonical: '/',
5      languages: {
6        'en-US': '/en-US',
7        'de-DE': '/de-DE',
8      },
9    },
10   openGraph: {
11     images: '/og-image.png',
12   },
13 };

```

<head> output

```

1 <link rel="canonical" href="https://acme.com" />
2 <link rel="alternate" hreflang="en-US" href="https://nextjs.org/en-US" />
3 <link rel="alternate" hreflang="de-DE" href="https://nextjs.org/de-DE" />
4 <meta property="og:image" content="https://acme.com/og-image.png" />

```

Good to know:

- `metadataBase` is typically set in root `app/layout.js` to apply to URL-based `metadata` fields across all routes.
- All URL-based `metadata` fields that require absolute URLs can be configured with a `metadataBase` option.
- `metadataBase` can contain a subdomain e.g. `https://app.acme.com` or base path e.g. `https://acme.com/start/from/here`
- If a `metadata` field provides an absolute URL, `metadataBase` will be ignored.
- Using a relative path in a URL-based `metadata` field without configuring a `metadataBase` will cause a build error.
- Next.js will normalize duplicate slashes between `metadataBase` (e.g. `https://acme.com/`) and a relative field (e.g. `/path`) to a single slash (e.g. `https://acme.com/path`)

Default value

If not configured, `metadataBase` has a **default value**

- When `VERCEL_URL` ↗ is detected: `https://${process.env.VERCEL_URL}` otherwise it falls back to `http://localhost:${process.env.PORT || 3000}`.

- When overriding the default, we recommend using environment variables to compute the URL. This allows configuring a URL for local development, staging, and production environments.

URL Composition

URL composition favors developer intent over default directory traversal semantics.

- Trailing slashes between `metadataBase` and `metadata` fields are normalized.
- An "absolute" path in a `metadata` field (that typically would replace the whole URL path) is treated as a "relative" path (starting from the end of `metadataBase`).

For example, given the following `metadataBase`:

```
TS app/api/layout.tsx
```

```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   metadataBase: new URL('https://acme.com/api'),
5 }
```

Any `metadata` fields that inherit the above `metadataBase` and set their own value will be resolved as follows:

<code>metadata</code> field	Resolved URL
/	<code>https://acme.com/api</code>
./	<code>https://acme.com/api</code>
payments	<code>https://acme.com/api/payments</code>
/payments	<code>https://acme.com/api/payments</code>
./payments	<code>https://acme.com/api/payments</code>
../payments	<code>https://acme.com/payments</code>
<code>https://beta.acme.com/api/payments</code>	<code>https://beta.acme.com/api/payments</code>

openGraph

```
JS layout.js / page.js
```

```
1 export const metadata = {
2   openGraph: {
```

```
3     title: 'Next.js',
4     description: 'The React Framework for the Web',
5     url: 'https://nextjs.org',
6     siteName: 'Next.js',
7     images: [
8         {
9             url: 'https://nextjs.org/og.png',
10            width: 800,
11            height: 600,
12        },
13        {
14            url: 'https://nextjs.org/og-alt.png',
15            width: 1800,
16            height: 1600,
17            alt: 'My custom alt',
18        },
19    ],
20    locale: 'en_US',
21    type: 'website',
22 },
23 };
```

📄 <head> output

```
1 <meta property="og:title" content="Next.js" />
2 <meta property="og:description" content="The React Framework for the Web" />
3 <meta property="og:url" content="https://nextjs.org/" />
4 <meta property="og:site_name" content="Next.js" />
5 <meta property="og:locale" content="en_US" />
6 <meta property="og:image:url" content="https://nextjs.org/og.png" />
7 <meta property="og:image:width" content="800" />
8 <meta property="og:image:height" content="600" />
9 <meta property="og:image:url" content="https://nextjs.org/og-alt.png" />
10 <meta property="og:image:width" content="1800" />
11 <meta property="og:image:height" content="1600" />
12 <meta property="og:image:alt" content="My custom alt" />
13 <meta property="og:type" content="website" />
```

💻 layout.js / page.js

```
1 export const metadata = {
2     openGraph: {
3         title: 'Next.js',
4         description: 'The React Framework for the Web',
5         type: 'article',
6         publishedTime: '2023-01-01T00:00:00.000Z',
7         authors: ['Seb', 'Josh'],
8     },
9 };
```

```
1 <meta property="og:title" content="Next.js" />
2 <meta property="og:description" content="The React Framework for the Web" />
3 <meta property="og:type" content="article" />
4 <meta property="article:published_time" content="2023-01-01T00:00:00.000Z" />
5 <meta property="article:author" content="Seb" />
6 <meta property="article:author" content="Josh" />
```

Good to know:

- It may be more convenient to use the [file-based Metadata API](#) for Open Graph images. Rather than having to sync the config export with actual files, the file-based API will automatically generate the correct metadata for you.

robots

```
1 export const metadata = {
2   robots: {
3     index: false,
4     follow: true,
5     nocache: true,
6     googleBot: {
7       index: true,
8       follow: false,
9       noimageindex: true,
10      'max-video-preview': -1,
11      'max-image-preview': 'large',
12      'max-snippet': -1,
13    },
14  },
15};
```

```
1 <meta name="robots" content="noindex, follow, nocache" />
2 <meta
3   name="googlebot"
4   content="index,nofollow,noimageindex,max-video-preview:-1,max-image-preview:large,
5 />
```

icons

Note: We recommend using the [file-based Metadata API](#) for icons where possible. Rather than having to sync the config export with actual files, the file-based API will automatically generate the correct metadata for you.

```

1  export const metadata = {
2    icons: {
3      icon: '/icon.png',
4      shortcut: '/shortcut-icon.png',
5      apple: '/apple-icon.png',
6      other: {
7        rel: 'apple-touch-icon-precomposed',
8        url: '/apple-touch-icon-precomposed.png',
9      },
10    },
11  };

```

```

1 <link rel="shortcut icon" href="/shortcut-icon.png" />
2 <link rel="icon" href="/icon.png" />
3 <link rel="apple-touch-icon" href="/apple-icon.png" />
4 <link
5   rel="apple-touch-icon-precomposed"
6   href="/apple-touch-icon-precomposed.png"
7 />

```

```

1  export const metadata = {
2    icons: {
3      icon: [{ url: '/icon.png' }, new URL('/icon.png', 'https://example.com')],
4      shortcut: ['/shortcut-icon.png'],
5      apple: [
6        { url: '/apple-icon.png' },
7        { url: '/apple-icon-x3.png', sizes: '180x180', type: 'image/png' },
8      ],
9      other: [
10        {
11          rel: 'apple-touch-icon-precomposed',
12          url: '/apple-touch-icon-precomposed.png',
13        },
14      ],
15    },
16  };

```

```

1 <link rel="shortcut icon" href="/shortcut-icon.png" />
2 <link rel="icon" href="/icon.png" />
3 <link rel="apple-touch-icon" href="/apple-icon.png" />

```

```
4 <link
5   rel="apple-touch-icon-precomposed"
6   href="/apple-touch-icon-precomposed.png"
7 />
8 <link rel="icon" href="https://example.com/icon.png" />
9 <link
10  rel="apple-touch-icon"
11  href="/apple-icon-x3.png"
12  sizes="180x180"
13  type="image/png"
14 />
```

Note: The `msapplication-*` meta tags are no longer supported in Chromium builds of Microsoft Edge, and thus no longer needed.

themeColor

Learn more about [theme-color ↗](#).

Simple theme color

`js` layout.js / page.js

```
1 export const metadata = {
2   themeColor: 'black',
3 };
```

`js` <head> output

```
<meta name="theme-color" content="black" />
```

With media attribute

`js` layout.js / page.js

```
1 export const metadata = {
2   themeColor: [
3     { media: '(prefers-color-scheme: light)', color: 'cyan' },
4     { media: '(prefers-color-scheme: dark)', color: 'black' },
5   ],
6 };
```

`js` <head> output

```
1 <meta name="theme-color" media="(prefers-color-scheme: light)" content="cyan" />
2 <meta name="theme-color" media="(prefers-color-scheme: dark)" content="black" />
```

manifest

A web application manifest, as defined in the [Web Application Manifest specification ↗](#).

 layout.js / page.js 

```
1 export const metadata = {
2   manifest: 'https://nextjs.org/manifest.json',
3 };
```

 <head> output 

```
<link rel="manifest" href="https://nextjs.org/manifest.json" />
```

twitter

Learn more about the [Twitter Card markup reference ↗](#).

 layout.js / page.js 

```
1 export const metadata = {
2   twitter: {
3     card: 'summary_large_image',
4     title: 'Next.js',
5     description: 'The React Framework for the Web',
6     siteId: '1467726470533754880',
7     creator: '@nextjs',
8     creatorId: '1467726470533754880',
9     images: ['https://nextjs.org/og.png'],
10   },
11 };
```

 <head> output 

```
1 <meta name="twitter:card" content="summary_large_image" />
2 <meta name="twitter:site:id" content="1467726470533754880" />
3 <meta name="twitter:creator" content="@nextjs" />
4 <meta name="twitter:creator:id" content="1467726470533754880" />
5 <meta name="twitter:title" content="Next.js" />
6 <meta name="twitter:description" content="The React Framework for the Web" />
7 <meta name="twitter:image" content="https://nextjs.org/og.png" />
```

js layout.js / page.js

```
1 export const metadata = {
2   twitter: {
3     card: 'app',
4     title: 'Next.js',
5     description: 'The React Framework for the Web',
6     siteId: '1467726470533754880',
7     creator: '@nextjs',
8     creatorId: '1467726470533754880',
9     images: {
10       url: 'https://nextjs.org/og.png',
11       alt: 'Next.js Logo',
12     },
13     app: {
14       name: 'twitter_app',
15       id: {
16         iphone: 'twitter_app://iphone',
17         ipad: 'twitter_app://ipad',
18         googleplay: 'twitter_app://googleplay',
19       },
20       url: {
21         iphone: 'https://iphone_url',
22         ipad: 'https://ipad_url',
23       },
24     },
25   },
26};
```

📄 <head> output

```
1 <meta name="twitter:site:id" content="1467726470533754880" />
2 <meta name="twitter:creator" content="@nextjs" />
3 <meta name="twitter:creator:id" content="1467726470533754880" />
4 <meta name="twitter:title" content="Next.js" />
5 <meta name="twitter:description" content="The React Framework for the Web" />
6 <meta name="twitter:card" content="app" />
7 <meta name="twitter:image" content="https://nextjs.org/og.png" />
8 <meta name="twitter:image:alt" content="Next.js Logo" />
9 <meta name="twitter:app:name:iphone" content="twitter_app" />
10 <meta name="twitter:app:id:iphone" content="twitter_app://iphone" />
11 <meta name="twitter:app:id:ipad" content="twitter_app://ipad" />
12 <meta name="twitter:app:id:googleplay" content="twitter_app://googleplay" />
```

```
13 <meta name="twitter:app:url:iphone" content="https://iphone_url" />
14 <meta name="twitter:app:url:ipad" content="https://ipad_url" />
15 <meta name="twitter:app:name:ipad" content="twitter_app" />
16 <meta name="twitter:app:name:googleplay" content="twitter_app" />
```

viewport

Note: The `viewport` meta tag is automatically set with the following default values. Usually, manual configuration is unnecessary as the default is sufficient. However, the information is provided for completeness.

js layout.js / page.js

```
1 export const metadata = {
2   viewport: {
3     width: 'device-width',
4     initialScale: 1,
5     maximumScale: 1,
6   },
7 }
```

html <head> output

```
1 <meta
2   name="viewport"
3   content="width=device-width, initial-scale=1, maximum-scale=1"
4 />
```

verification

js layout.js / page.js

```
1 export const metadata = {
2   verification: {
3     google: 'google',
4     yandex: 'yandex',
5     yahoo: 'yahoo',
6     other: {
7       me: ['my-email', 'my-link'],
8     },
9   },
10 }
```

html <head> output

```
1 <meta name="google-site-verification" content="google" />
2 <meta name="y_key" content="yahoo" />
3 <meta name="yandex-verification" content="yandex" />
4 <meta name="me" content="my-email" />
5 <meta name="me" content="my-link" />
```

appleWebApp

js layout.js / page.js

```
1 export const metadata = {
2   itunes: {
3     appId: 'myAppStoreID',
4     appArgument: 'myAppArgument',
5   },
6   appleWebApp: {
7     title: 'Apple Web App',
8     statusBarStyle: 'black-translucent',
9     startupImage: [
10       '/assets/startup/apple-touch-startup-image-768x1004.png',
11       {
12         url: '/assets/startup/apple-touch-startup-image-1536x2008.png',
13         media: '(device-width: 768px) and (device-height: 1024px)',
14       },
15     ],
16   },
17};
```

□ <head> output

```
1 <meta
2   name="apple-itunes-app"
3   content="app-id=myAppStoreID, app-argument=myAppArgument"
4 />
5 <meta name="apple-mobile-web-app-capable" content="yes" />
6 <meta name="apple-mobile-web-app-title" content="Apple Web App" />
7 <link
8   href="/assets/startup/apple-touch-startup-image-768x1004.png"
9   rel="apple-touch-startup-image"
10 />
11 <link
12   href="/assets/startup/apple-touch-startup-image-1536x2008.png"
13   media="(device-width: 768px) and (device-height: 1024px)"
14   rel="apple-touch-startup-image"
15 />
16 <meta
17   name="apple-mobile-web-app-status-bar-style"
18   content="black-translucent"
19 />
```

alternates

js layout.js / page.js

```
1 export const metadata = {
2   alternates: {
3     canonical: 'https://nextjs.org',
4     languages: {
5       'en-US': 'https://nextjs.org/en-US',
6       'de-DE': 'https://nextjs.org/de-DE',
7     },
8     media: {
9       'only screen and (max-width: 600px)': 'https://nextjs.org/mobile',
10    },
11    types: {
12      'application/rss+xml': 'https://nextjs.org/rss',
13    },
14  },
15};
```

□ <head> output

```
1 <link rel="canonical" href="https://nextjs.org" />
2 <link rel="alternate" hreflang="en-US" href="https://nextjs.org/en-US" />
3 <link rel="alternate" hreflang="de-DE" href="https://nextjs.org/de-DE" />
4 <link
5   rel="alternate"
6   media="only screen and (max-width: 600px)"
7   href="https://nextjs.org/mobile"
8 />
9 <link
10  rel="alternate"
11  type="application/rss+xml"
12  href="https://nextjs.org/rss"
13 />
```

appLinks

js layout.js / page.js

```
1 export const metadata = {
2   appLinks: {
3     ios: {
4       url: 'https://nextjs.org/ios',
5       app_store_id: 'app_store_id',
6     },
7     android: {
8       package: 'com.example.android/package',
9       app_name: 'app_name_android',
```

```
10     },
11   web: {
12     url: 'https://nextjs.org/web',
13     should_fallback: true,
14   },
15 },
16 };
```

📄 <head> output



```
1 <meta property="al:ios:url" content="https://nextjs.org/ios" />
2 <meta property="al:ios:app_store_id" content="app_store_id" />
3 <meta property="al:android:package" content="com.example.android/package" />
4 <meta property="al:android:app_name" content="app_name_android" />
5 <meta property="al:web:url" content="https://nextjs.org/web" />
6 <meta property="al:web:should_fallback" content="true" />
```

archives

Describes a collection of records, documents, or other materials of historical interest ([source ↗](#)).

JS layout.js / page.js



```
1 export const metadata = {
2   archives: ['https://nextjs.org/13'],
3 };
```

📄 <head> output



```
<link rel="archives" href="https://nextjs.org/13" />
```

assets

JS layout.js / page.js



```
1 export const metadata = {
2   assets: ['https://nextjs.org/assets'],
3 };
```

📄 <head> output



```
<link rel="assets" href="https://nextjs.org/assets" />
```

bookmarks

js layout.js / page.js

```
1 export const metadata = {  
2   bookmarks: ['https://nextjs.org/13'],  
3 };
```

📄 <head> output

```
<link rel="bookmarks" href="https://nextjs.org/13" />
```

category

js layout.js / page.js

```
1 export const metadata = {  
2   category: 'technology',  
3 };
```

📄 <head> output

```
<meta name="category" content="technology" />
```

other

All metadata options should be covered using the built-in support. However, there may be custom metadata tags specific to your site, or brand new metadata tags just released. You can use the `other` option to render any custom metadata tag.

js layout.js / page.js

```
1 export const metadata = {  
2   other: {  
3     custom: 'meta',  
4   },  
5 };
```

📄 <head> output

```
<meta name="custom" content="meta" />
```

Unsupported Metadata

The following metadata types do not currently have built-in support. However, they can still be rendered in the layout or page itself.

Metadata	Recommendation
<meta http-equiv="...">	Use appropriate HTTP Headers via redirect() , Middleware , Security Headers
<base>	Render the tag in the layout or page itself.
<noscript>	Render the tag in the layout or page itself.
<style>	Learn more about styling in Next.js .
<script>	Learn more about using scripts .
<link rel="stylesheet" />	<code>import</code> stylesheets directly in the layout or page itself.
<link rel="preload" />	Use ReactDOM preload method
<link rel="preconnect" />	Use ReactDOM preconnect method
<link rel="dns-prefetch" />	Use ReactDOM prefetchDNS method

Resource hints

The `<link>` element has a number of `rel` keywords that can be used to hint to the browser that a external resource is likely to be needed. The browser uses this information to apply preloading optimizations depending on the keyword.

While the Metadata API doesn't directly support these hints, you can use new [ReactDOM methods ↗](#) to safely insert them into the `<head>` of the document.

```
TS app/preload-resources.tsx
```

```
1  'use client';
2
3  import ReactDOM from 'react-dom';
4
5  export function PreloadResources() {
```

```
6   ReactDOM.preload('...', { as: '...' });
7   ReactDOM.preconnect('...', { crossOrigin: '...' });
8   ReactDOM.prefetchDNS('...');
9
10  return null;
11 }
```

<link rel="preload">

Start loading a resource early in the page rendering (browser) lifecycle. [MDN Docs ↗](#).

```
ReactDOM.preload(href: string, options: { as: string })
```

 <head> output



```
<link rel="preload" href="..." as="..." />
```

<link rel="preconnect">

Preemptively initiate a connection to an origin. [MDN Docs ↗](#).

```
ReactDOM.preconnect(href: string, options?: { crossOrigin?: string })
```

 <head> output



```
<link rel="preconnect" href="..." crossorigin />
```

<link rel="dns-prefetch">

Attempt to resolve a domain name before resources get requested. [MDN Docs ↗](#).

```
ReactDOM.prefetchDNS(href: string)
```

 <head> output



```
<link rel="dns-prefetch" href="..." />
```

Good to know:

- These methods are currently only supported in Client Components.
 - Note: Client Components are still Server Side Rendered on initial page load.

- Next.js in-built features such as `next/font`, `next/image` and `next/script` automatically handle relevant resource hints.
- React 18.3 does not yet include type definitions for `ReactDOM.preload`, `ReactDOM.preconnect`, and `ReactDOM.preconnectDNS`. You can use `// @ts-ignore` as a temporary solution to avoid type errors.

Types

You can add type safety to your metadata by using the `Metadata` type. If you are using the [built-in TypeScript plugin](#) in your IDE, you do not need to manually add the type, but you can still explicitly add it if you want.

`metadata` object

```
1 import type { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: 'Next.js',
5 }
```

`generateMetadata` function

Regular function

```
1 import type { Metadata } from 'next';
2
3 export function generateMetadata(): Metadata {
4   return {
5     title: 'Next.js',
6   };
7 }
```

Async function

```
1 import type { Metadata } from 'next';
2
3 export async function generateMetadata(): Promise<Metadata> {
4   return {
5     title: 'Next.js',
6   };
7 }
```

With segment props

```
1 import type { Metadata } from 'next';
2
3 type Props = {
4   params: { id: string };
5   searchParams: { [key: string]: string | string[] | undefined };
6 };
7
8 export function generateMetadata({ params, searchParams }: Props): Metadata {
9   return {
10     title: 'Next.js',
11   };
12 }
13
14 export default function Page({ params, searchParams }: Props) {}
```

With parent metadata

```
1 import type { Metadata, ResolvingMetadata } from 'next';
2
3 export async function generateMetadata(
4   { params, searchParams }: Props,
5   parent: ResolvingMetadata,
6 ): Promise<Metadata> {
7   return {
8     title: 'Next.js',
9   };
10 }
```

JavaScript Projects

For JavaScript projects, you can use JSDoc to add type safety.

```
1 /** @type {import("next").Metadata} */
2 export const metadata = {
3   title: 'Next.js',
4 };
```

Next Steps

View all the Metadata API options.

App Router > ... > Optimizing

Metadata

Use the Metadata API to define metadata in any layout or page.

> Menu

App Router > ... > Functions > generateStaticParams

generateStaticParams

The `generateStaticParams` function can be used in combination with [dynamic route segments](#) to [statically generate](#) routes at build time instead of on-demand at request time.

js app/blog/[slug]/page.js

```
1 // Return a list of `params` to populate the [slug] dynamic segment
2 export async function generateStaticParams() {
3   const posts = await fetch('https://.../posts').then((res) => res.json());
4
5   return posts.map((post) => ({
6     slug: post.slug,
7   }));
8 }
9
10 // Multiple versions of this page will be statically generated
11 // using the `params` returned by `generateStaticParams`
12 export default function Page({ params }) {
13   const { slug } = params;
14   // ...
15 }
```

Good to know

- You can use the `dynamicParams` segment config option to control what happens when a dynamic segment is visited that was not generated with `generateStaticParams`.
- During `next dev`, `generateStaticParams` will be called when you navigate to a route.
- During `next build`, `generateStaticParams` runs before the corresponding Layouts or Pages are generated.
- During revalidation (ISR), `generateStaticParams` will not be called again.
- `generateStaticParams` replaces the `getStaticPaths` function in the Pages Router.

Parameters

`options.params` (optional)

If multiple dynamic segments in a route use `generateStaticParams`, the child `generateStaticParams` function is executed once for each set of `params` the parent generates.

The `params` object contains the populated `params` from the parent `generateStaticParams`, which can be used to [generate the `params` in a child segment](#).

Returns

`generateStaticParams` should return an array of objects where each object represents the populated dynamic segments of a single route.

- Each property in the object is a dynamic segment to be filled in for the route.
- The properties name is the segment's name, and the properties value is what that segment should be filled in with.

Example Route

`generateStaticParams` Return Type

<code>/product/[id]</code>	<code>{ id: string }[]</code>
<code>/products/[category]/[product]</code>	<code>{ category: string, product: string }[]</code>
<code>/products/[...slug]</code>	<code>{ slug: string[] }[]</code>

Single Dynamic Segment

ts app/product/[id]/page.tsx

```
1 export function generateStaticParams() {
2   return [{ id: '1' }, { id: '2' }, { id: '3' }];
3 }
4
5 // Three versions of this page will be statically generated
6 // using the `params` returned by `generateStaticParams`
7 // - /product/1
8 // - /product/2
9 // - /product/3
10 export default function Page({ params }: { params: { id: string } }) {
11   const { id } = params;
```

```
12 // ...
13 }
```

Multiple Dynamic Segments

TS app/products/[category]/[product]/page.tsx

```
1 export function generateStaticParams() {
2   return [
3     { category: 'a', product: '1' },
4     { category: 'b', product: '2' },
5     { category: 'c', product: '3' },
6   ];
7 }
8
9 // Three versions of this page will be statically generated
10 // using the `params` returned by `generateStaticParams`
11 // - /product/a/1
12 // - /product/b/2
13 // - /product/c/3
14 export default function Page({
15   params,
16 }: {
17   params: { category: string; product: string };
18 }) {
19   const { category, product } = params;
20   // ...
21 }
```

Catch-all Dynamic Segment

TS app/product/...slug/page.tsx

```
1 export function generateStaticParams() {
2   return [{ slug: ['a', '1'] }, { slug: ['b', '2'] }, { slug: ['c', '3'] }];
3 }
4
5 // Three versions of this page will be statically generated
6 // using the `params` returned by `generateStaticParams`
7 // - /product/a/1
8 // - /product/b/2
9 // - /product/c/3
10 export default function Page({ params }: { params: { slug: string[] } }) {
```

```
11   const { slug } = params;
12   // ...
13 }
```

Examples

Multiple Dynamic Segments in a Route

You can generate params for dynamic segments above the current layout or page, but **not below**. For example, given the `app/products/[category]/[product]` route:

- `app/products/[category]/[product]/page.js` can generate params for **both** `[category]` and `[product]`.
- `app/products/[category]/layout.js` can **only** generate params for `[category]`.

There are two approaches to generating params for a route with multiple dynamic segments:

Generate params from the bottom up

Generate multiple dynamic segments from the child route segment.

```
TS app/products/[category]/[product]/page.tsx
```

```
1 // Generate segments for both [category] and [product]
2 export async function generateStaticParams() {
3   const products = await fetch('https://.../products').then((res) =>
4     res.json(),
5   );
6
7   return products.map((product) => ({
8     category: product.category.slug,
9     product: product.id,
10    }));
11 }
12
13 export default function Page({
14   params,
15 }: {
16   params: { category: string; product: string };
17 }) {
18   // ...
19 }
```

Generate params from the top down

Generate the parent segments first and use the result to generate the child segments.

TS app/products/[category]/layout.tsx

```
1 // Generate segments for [category]
2 export async function generateStaticParams() {
3   const products = await fetch('https://.../products').then((res) =>
4     res.json(),
5   );
6
7   return products.map((product) => ({
8     category: product.category.slug,
9   }));
10 }
11
12 export default function Layout({ params }: { params: { category: string } }) {
13   // ...
14 }
```

A child route segment's `generateStaticParams` function is executed once for each segment a parent `generateStaticParams` generates.

The child `generateStaticParams` function can use the `params` returned from the parent `generateStaticParams` function to dynamically generate its own segments.

TS app/products/[category]/[product]/page.tsx

```
1 // Generate segments for [product] using the `params` passed from
2 // the parent segment's `generateStaticParams` function
3 export async function generateStaticParams({
4   params: { category },
5 }: {
6   params: { category: string };
7 }) {
8   const products = await fetch(
9     `https://.../products?category=${category}`,
10   ).then((res) => res.json());
11
12   return products.map((product) => ({
13     product: product.id,
14   }));
15 }
16
17 export default function Page({
18   params,
19 }: {
20   params: { category: string; product: string };
21 }) {
```

```
22    // ...
23 }
```

Good to know: When rendering a route, Next.js will automatically deduplicate `fetch` requests for the same data across `generateMetadata`, `generateStaticParams`, Layouts, Pages, and Server Components. React `cache` can be used if `fetch` is unavailable.

> Menu

App Router > ... > Functions > headers

headers

The `headers` function allows you to read the HTTP incoming request headers from a [Server Component](#).

headers()

This API extends the [Web Headers API ↗](#). It is **read-only**, meaning you cannot `set` / `delete` the outgoing request headers.

TS app/page.tsx

```
1 import { headers } from 'next/headers';
2
3 export default function Page() {
4   const headersList = headers();
5   const referer = headersList.get('referer');
6
7   return <div>Referer: {referer}</div>;
8 }
```

Good to know:

- `headers()` is a [Dynamic Function](#) whose returned values cannot be known ahead of time. Using it in a layout or page will opt a route into [dynamic rendering](#) at request time.

API Reference

```
const headersList = headers();
```

Parameters

`headers` does not take any parameters.

Returns

`headers` returns a **read-only** [Web Headers](#) object.

- `Headers.entries()` ↗: Returns an [iterator](#) ↗ allowing to go through all key/value pairs contained in this object.
- `Headers.forEach()` ↗: Executes a provided function once for each key/value pair in this `Headers` object.
- `Headers.get()` ↗: Returns a [String](#) ↗ sequence of all the values of a header within a `Headers` object with a given name.
- `Headers.has()` ↗: Returns a boolean stating whether a `Headers` object contains a certain header.
- `Headers.keys()` ↗: Returns an [iterator](#) ↗ allowing you to go through all keys of the key/value pairs contained in this object.
- `Headers.values()` ↗: Returns an [iterator](#) ↗ allowing you to go through all values of the key/value pairs contained in this object.

Examples

Usage with Data Fetching

`headers()` can be used in combination with [Suspense for Data Fetching](#).

JS app/page.js

```
1 import { headers } from 'next/headers';
2
3 async function getUser() {
4   const headersInstance = headers();
5   const authorization = headersInstance.get('authorization');
6   // Forward the authorization header
7   const res = await fetch('...', {
8     headers: { authorization },
9   });
10  return res.json();
11 }
12
13 export default async function UserPage() {
14   const user = await getUser();
15   return <h1>{user.name}</h1>;
16 }
```

> Menu

App Router > ... > Functions > ImageResponse

ImageResponse

The `ImageResponse` constructor allows you to generate dynamic images using JSX and CSS. This is useful for generating social media images such as Open Graph images, Twitter cards, and more.

The following options are available for `ImageResponse`:

```
1 import { ImageResponse } from 'next/server'
2
3 new ImageResponse(
4   element: ReactElement,
5   options: {
6     width?: number = 1200
7     height?: number = 630
8     emoji?: 'twemoji' | 'blobemoji' | 'noto' | 'openemoji' = 'twemoji',
9     fonts?: {
10       name: string,
11       data: ArrayBuffer,
12       weight: number,
13       style: 'normal' | 'italic'
14     }[]
15     debug?: boolean = false
16
17     // Options that will be passed to the HTTP response
18     status?: number = 200
19     statusText?: string
20     headers?: Record<string, string>
21   },
22 )
```

Supported CSS Properties

Please refer to [Satori's documentation](#) for a list of supported HTML and CSS features.

> Menu

App Router > ... > Functions > NextRequest

NextRequest

NextRequest extends the [Web Request API ↗](#) with additional convenience methods.

cookies

Read or mutate the [Set-Cookie ↗](#) header of the request.

set(name, value)

Given a name, set a cookie with the given value on the request.

```
1 // Given incoming request /home
2 // Set a cookie to hide the banner
3 // request will have a `Set-Cookie:show-banner=false;path=/home` header
4 request.cookies.set('show-banner', 'false');
```

get(name)

Given a cookie name, return the value of the cookie. If the cookie is not found, `undefined` is returned. If multiple cookies are found, the first one is returned.

```
1 // Given incoming request /home
2 // { name: 'show-banner', value: 'false', Path: '/home' }
3 request.cookies.get('show-banner');
```

getAll()

Given a cookie name, return the values of the cookie. If no name is given, return all cookies on the request.

```
1 // Given incoming request /home
2 // [
3 //   { name: 'experiments', value: 'new-pricing-page', Path: '/home' },
4 //   { name: 'experiments', value: 'winter-launch', Path: '/home' },
5 // ]
6 request.cookies.getAll('experiments');
7 // Alternatively, get all cookies for the request
8 request.cookies.getAll();
```

delete(name)

Given a cookie name, delete the cookie from the request.

```
1 // Returns true for deleted, false is nothing is deleted
2 request.cookies.delete('experiments');
```

has(name)

Given a cookie name, return `true` if the cookie exists on the request.

```
1 // Returns true if cookie exists, false if it does not
2 request.cookies.has('experiments');
```

clear()

Remove the `Set-Cookie` header from the request.

```
request.cookies.clear();
```

nextUrl

Extends the native [URL](#) API with additional convenience methods, including Next.js specific properties.

```
1 // Given a request to /home, pathname is /home
2 request.nextUrl.pathname;
3 // Given a request to /home?name=lee, searchParams is { 'name': 'lee' }
4 request.nextUrl.searchParams;
```


> Menu

App Router > ... > Functions > NextResponse

NextResponse

NextResponse extends the [Web Response API ↗](#) with additional convenience methods.

cookies

Read or mutate the [Set-Cookie ↗](#) header of the response.

set(name, value)

Given a name, set a cookie with the given value on the response.

```
1 // Given incoming request /home
2 let response = NextResponse.next();
3 // Set a cookie to hide the banner
4 response.cookies.set('show-banner', 'false');
5 // Response will have a `Set-Cookie:show-banner=false;path=/home` header
6 return response;
```

get(name)

Given a cookie name, return the value of the cookie. If the cookie is not found, `undefined` is returned. If multiple cookies are found, the first one is returned.

```
1 // Given incoming request /home
2 let response = NextResponse.next();
3 // { name: 'show-banner', value: 'false', Path: '/home' }
4 response.cookies.get('show-banner');
```

getAll()

Given a cookie name, return the values of the cookie. If no name is given, return all cookies on the response.

```
1 // Given incoming request /home
2 let response = NextResponse.next();
3 //
4 // { name: 'experiments', value: 'new-pricing-page', Path: '/home' },
5 // { name: 'experiments', value: 'winter-launch', Path: '/home' },
6 //
7 response.cookies.getAll('experiments');
8 // Alternatively, get all cookies for the response
9 response.cookies.getAll();
```

delete(name)

Given a cookie name, delete the cookie from the response.

```
1 // Given incoming request /home
2 let response = NextResponse.next();
3 // Returns true for deleted, false is nothing is deleted
4 response.cookies.delete('experiments');
```

json()

Produce a response with the given JSON body.

app/api/route.ts

```
1 import { NextResponse } from 'next/server';
2
3 export async function GET(request: Request) {
4   return NextResponse.json({ error: 'Internal Server Error' }, { status: 500 });
5 }
```

redirect()

Produce a response that redirects to a [URL ↗](#).

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

```
2
3 return NextResponse.redirect(new URL('/new', request.url));
```

The `URL` can be created and modified before being used in the `NextResponse.redirect()` method. For example, you can use the `request.nextUrl` property to get the current URL, and then modify it to redirect to a different URL.

```
1 import { NextResponse } from 'next/server';
2
3 // Given an incoming request...
4 const loginUrl = new URL('/login', request.url);
5 // Add ?from=/incoming-url to the /login URL
6 loginUrl.searchParams.set('from', request.nextUrl.pathname);
7 // And redirect to the new URL
8 return NextResponse.redirect(loginUrl);
```

rewrite()

Produce a response that rewrites (proxies) the given `URL` while preserving showing the original URL.

```
1 import { NextResponse } from 'next/server';
2
3 // Incoming request: /about, browser shows /about
4 // Rewritten request: /proxy, browser shows /about
5 return NextResponse.rewrite(new URL('/proxy', request.url));
```

next()

The `next()` method is useful for Middleware, as it allows you to return early and continue routing.

```
1 import { NextResponse } from 'next/server';
2
3 return NextResponse.next();
```

You can also forward `headers` when producing the response:

```
1 import { NextResponse } from 'next/server';
2
```

```
3 // Given an incoming request...
4 const newHeaders = new Headers(request.headers);
5 // Add a new header
6 newHeaders.set('x-version', '123');
7 // And produce a response with the new headers
8 return NextResponse.next({
9   request: {
10     // New request headers
11     headers: newHeaders,
12   },
13 });
```

> Menu

App Router > ... > Functions > notFound

notFound

The `notFound` function allows you to render the `not-found file` within a route segment as well as inject a `<meta name="robots" content="noindex" />` tag.

notFound()

Invoking the `notFound()` function throws a `NEXT_NOT_FOUND` error and terminates rendering of the route segment in which it was thrown. Specifying a `not-found file` allows you to gracefully handle such errors by rendering a Not Found UI within the segment.

JS app/user/[id]/page.js



```
1 import { notFound } from 'next/navigation';
2
3 async function fetchUsers(id) {
4   const res = await fetch('https://...');
5   if (!res.ok) return undefined;
6   return res.json();
7 }
8
9 export default async function Profile({ params }) {
10   const user = await fetchUser(params.id);
11
12   if (!user) {
13     notFound();
14   }
15
16   // ...
17 }
```

Note: `notFound()` does not require you to use `return notFound()` due to using the TypeScript `never ↗` type.

> Menu

App Router > ... > Functions > redirect

redirect

The `redirect` function allows you to redirect the user to another URL. If you need to redirect to a 404, you can use the `notFound` function.

redirect()

Invoking the `redirect()` function throws a `NEXT_REDIRECT` error and terminates rendering of the route segment in which it was thrown. `redirect()` can be called with a relative or an absolute URL.

JS app/team/[id]/page.js

```
1 import { redirect } from 'next/navigation';
2
3 async function fetchTeam(id) {
4   const res = await fetch(`https://...`);
5   if (!res.ok) return undefined;
6   return res.json();
7 }
8
9 export default async function Profile({ params }) {
10   const team = await fetchTeam(params.id);
11   if (!team) {
12     redirect('/login');
13   }
14
15   // ...
16 }
```

Note: `redirect()` does not require you to use `return redirect()` due to using the TypeScript `never ↗` type.

> Menu

App Router > ... > Functions > revalidatePath

revalidatePath

`revalidatePath` allows you to revalidate data associated with a specific path. This is useful for scenarios where you want to update your cached data without waiting for a revalidation period to expire.

TS app/api/revalidate/route.ts

```
1 import { NextRequest, NextResponse } from 'next/server';
2 import { revalidatePath } from 'next/cache';
3
4 export async function GET(request: NextRequest) {
5   const path = request.nextUrl.searchParams.get('path') || '/';
6   revalidatePath(path);
7   return NextResponse.json({ revalidated: true, now: Date.now() });
8 }
```

Good to know:

- `revalidatePath` is available in both [Node.js](#) and [Edge runtimes](#).
- `revalidatePath` will revalidate *all segments* under a dynamic route segment. For example, if you have a dynamic segment `/product/[id]` and you call `revalidatePath('/product/[id]')`, then all segments under `/product/[id]` will be revalidated as requested.
- `revalidatePath` only invalidates the cache when the path is next visited. This means calling `revalidatePath` with a dynamic route segment will not immediately trigger many revalidations at once. The invalidation only happens when the path is next visited.

Parameters

`revalidatePath(path: string): void;`

- `path`: A string representing the filesystem path associated with the data you want to revalidate. This is **not** the literal route segment (e.g. `/product/123`) but instead the path on the filesystem (e.g. `/product/[id]`).

Returns

`revalidatePath` does not return any value.

Examples

Node.js Runtime

app/api/revalidate/route.ts

```
1 import { NextRequest, NextResponse } from 'next/server';
2 import { revalidatePath } from 'next/cache';
3
4 export async function GET(request: NextRequest) {
5   const path = request.nextUrl.searchParams.get('path') || '/';
6   revalidatePath(path);
7   return NextResponse.json({ revalidated: true, now: Date.now() });
8 }
```

Edge Runtime

app/api/revalidate/route.ts

```
1 import { NextRequest, NextResponse } from 'next/server';
2 import { revalidatePath } from 'next/cache';
3
4 export const runtime = 'edge';
5
6 export async function GET(request: NextRequest) {
7   const path = request.nextUrl.searchParams.get('path') || '/';
8   revalidatePath(path);
9   return NextResponse.json({ revalidated: true, now: Date.now() });
10 }
```

> Menu

App Router > ... > Functions > revalidateTag

revalidateTag

`revalidateTag` allows you to revalidate data associated with a specific cache tag. This is useful for scenarios where you want to update your cached data without waiting for a revalidation period to expire.

TS app/api/revalidate/route.ts

```
1 import { NextRequest, NextResponse, revalidateTag } from 'next/server';
2
3 export async function GET(request: NextRequest) {
4   const tag = request.nextUrl.searchParams.get('tag');
5   revalidateTag(tag);
6   return NextResponse.json({ revalidated: true, now: Date.now() });
7 }
```

Good to know:

- `revalidateTag` is available in both Node.js and Edge runtimes.

Parameters

`revalidateTag(tag: string): void;`

- `tag`: A string representing the cache tag associated with the data you want to revalidate.

You can add tags to `fetch` as follows:

`fetch(url, { next: { tags: [...] } });`

Returns

`revalidateTag` does not return any value.

Examples

Node.js Runtime

TS app/api/revalidate/route.ts

```
1 import { NextRequest, NextResponse } from 'next/server';
2 import { revalidateTag } from 'next/cache';
3
4 export async function GET(request: NextRequest) {
5   const tag = request.nextUrl.searchParams.get('tag');
6   revalidateTag(tag);
7   return NextResponse.json({ revalidated: true, now: Date.now() });
8 }
```

Edge Runtime

TS app/api/revalidate/route.ts

```
1 import { NextRequest, NextResponse } from 'next/server';
2 import { revalidateTag } from 'next/cache';
3
4 export const runtime = 'edge';
5
6 export async function GET(request: NextRequest) {
7   const tag = request.nextUrl.searchParams.get('tag');
8   revalidateTag(tag);
9   return NextResponse.json({ revalidated: true, now: Date.now() });
10 }
```

> Menu

App Router > ... > Functions > useParams

useParams

`useParams` is a **Client Component** hook that lets you read a route's [dynamic params](#) filled in by the current URL.

Note: Support for `useParams` is added in Next.js 13.3.

TS app/example-client-component.tsx

```
1 'use client';
2
3 import { useParams } from 'next/navigation';
4
5 export default function ExampleClientComponent() {
6   const params = useParams();
7
8   // Route -> /shop/[tag]/[item]
9   // URL -> /shop/shoes/nike-air-max-97
10  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
11  console.log(params);
12
13  return <></>;
14 }
```

Parameters

```
const params = useParams();
```

`useParams` does not take any parameters.

Returns

`useParams` returns an object containing the current route's filled in [dynamic parameters](#).

- Each property in the object is an active dynamic segment.
- The properties name is the segment's name, and the properties value is what the segment is filled in with.
- The properties value will either be a `string` or array of `string`'s depending on the [type of dynamic segment](#).
- If the route contains no dynamic parameters, `useParams` returns an empty object.
- If used in `pages`, `useParams` will return `null`.

For example:

Route	URL	<code>useParams()</code>
<code>app/shop/page.js</code>	<code>/shop</code>	<code>null</code>
<code>app/shop/[slug]/page.js</code>	<code>/shop/1</code>	<code>{ slug: '1' }</code>
<code>app/shop/[tag]/[item]/page.js</code>	<code>/shop/1/2</code>	<code>{ tag: '1', item: '2' }</code>
<code>app/shop/...slug/page.js</code>	<code>/shop/1/2</code>	<code>{ slug: ['1', '2'] }</code>

> Menu

App Router > ... > Functions > usePathname

usePathname

usePathname is a **Client Component** hook that lets you read the current URL's **pathname**.

TS app/example-client-component.tsx

```
1 'use client';
2
3 import { usePathname } from 'next/navigation';
4
5 export default function ExampleClientComponent() {
6   const pathname = usePathname();
7   return <>Current pathname: {pathname}</>;
8 }
```

Good to know:

- usePathname is a **Client Component** hook and is **not supported** in **Server Components**.
- Compatibility mode:
 - usePathname can return null when a **fallback route** is being rendered or when a **pages** directory page has been **automatically statically optimized** by Next.js and the router is not ready.
 - Next.js will automatically update your types if it detects both an **app** and **pages** directory in your project.

Parameters

```
const pathname = usePathname();
```

usePathname does not take any parameters.

Returns

`usePathname` returns a string of the current URL's pathname. For example:

URL	Returned value
/	'/'
/dashboard	'/dashboard'
/dashboard?v=2	'/dashboard'
/blog/hello-world	'/blog/hello-world'

Examples

Do something in response to a route change

TS app/example-client-component.tsx

```
1  'use client';
2
3  import { usePathname, useSearchParams } from 'next/navigation';
4
5  function ExampleClientComponent() {
6    const pathname = usePathname();
7    const searchParams = useSearchParams();
8    useEffect(() => {
9      // Do something here...
10    }, [pathname, searchParams]);
11 }
```

> Menu

App 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 app/_components/web-vitals.js

```
1 'use client';
2
3 import { useReportWebVitals } from 'next/web-vitals';
4
5 export function WebVitals() {
6   useReportWebVitals((metric) => {
7     console.log(metric);
8   });
9 }
```

JS app/layout.js

```
1 import { WebVitals } from './_components/web-vitals';
2
3 export default function Layout({ children }) {
4   return (
5     <html>
6       <body>
7         <WebVitals />
8         {children}
9       </body>
10      </html>
11    );
12 }
```

Since the `useReportWebVitals` hook requires the `"use client"` directive, the most performant approach is to create a separate component that the root layout imports. This confines the client boundary exclusively to the `WebVitals` component.

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](#)s.
-

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.

app/components/web-vitals.tsx

```
1  'use client';
2
3  import { useReportWebVitals } from 'next/web-vitals';
4
5  export function WebVitals() {
6    useReportWebVitals((metric) => {
7      switch (metric.name) {
8        case 'FCP': {
9          // handle FCP results
10       }
11        case 'LCP': {
12          // handle LCP results
13        }
14        // ...
15      }
16    });
17 }
```

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
4   window.gtag('event', metric.name, {
5     value: Math.round(metric.name === 'CLS' ? metric.value * 1000 : metric.value), //
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

App Router > ... > Functions > useRouter

useRouter

The `useRouter` hook allows you to programmatically change routes inside [Client Components](#).

Recommendation: Use the [`<Link>` component](#) for navigation unless you have a specific requirement for using `useRouter`.

ts app/example-client-component.tsx

```
1  'use client';
2
3  import { useRouter } from 'next/navigation';
4
5  export default function Page() {
6    const router = useRouter();
7
8    return (
9      <button type="button" onClick={() => router.push('/dashboard')}>
10        Dashboard
11      </button>
12    );
13 }
```

useRouter()

- `router.push(href: string)`: Perform a client-side navigation to the provided route. Adds a new entry into the [browser's history](#) ↗ stack.
- `router.replace(href: string)`: Perform a client-side navigation to the provided route without adding a new entry into the [browser's history stack](#) ↗ .
- `router.refresh()`: Refresh the current route. Making a new request to the server, re-fetching data requests, and re-rendering Server Components. The client will merge the updated React Server

Component payload without losing unaffected client-side React (e.g. `useState`) or browser state (e.g. scroll position).

- `router.prefetch(href: string)`: [Prefetch](#) the provided route for faster client-side transitions.
- `router.back()`: Navigate back to the previous route in the browser's history stack using [soft navigation](#).
- `router.forward()`: Navigate forwards to the next page in the browser's history stack using [soft navigation](#).

Good to know:

- The `push()` and `replace()` methods will perform a [soft navigation](#) if the new route has been prefetched, and either, doesn't include [dynamic segments](#) or has the same dynamic parameters as the current route.
- `next/link` automatically prefetch routes as they become visible in the viewport.
- `refresh()` could re-produce the same result if fetch requests are cached. Other dynamic functions like `cookies` and `headers` could also change the response.

Migrating from the `pages` directory:

- The new `useRouter` hook should be imported from `next/navigation` and not `next/router`
- The `pathname` string has been removed and is replaced by `usePathname()`
- The `query` object has been removed and is replaced by `useSearchParams()`
- `router.events` is not currently supported. [See below](#).

[View the full migration guide.](#)

Examples

Router Events

You can listen for page changes by composing other Client Component hooks like `usePathname` and `useSearchParams`.

 app/components/navigation-events.js



```
1 'use client';
2
3 import { useEffect } from 'react';
4 import { usePathname, useSearchParams } from 'next/navigation';
5
6 export function NavigationEvents() {
7   const pathname = usePathname();
```

```
8  const searchParams = useSearchParams();
9
10 useEffect(() => {
11   const url = pathname + searchParams.toString();
12   console.log(url);
13   // You can now use the current URL
14   // ...
15 }, [pathname, searchParams]);
16
17 return null;
18 }
```

Which can be imported into a layout.

Js app/layout.js

```
1 import { Suspense } from 'react';
2 import { NavigationEvents } from './components/navigation-events';
3
4 export default function Layout({ children }) {
5   return (
6     <html lang="en">
7       <body>
8         {children}
9
10        <Suspense fallback={null}>
11          <NavigationEvents />
12        </Suspense>
13      </body>
14    </html>
15  );
16}
```

Note: `<NavigationEvents>` is wrapped in a `Suspense` boundary because `useSearchParams()` causes client-side rendering up to the closest `Suspense` boundary during static rendering. [Learn more](#).

> Menu

App Router > ... > Functions > useSearchParams

useSearchParams

useSearchParams is a **Client Component** hook that lets you read the current URL's **query string**.

useSearchParams returns a **read-only** version of the [URLSearchParams](#) interface.

ts app/dashboard/search-bar.tsx

```
1  'use client';
2
3  import { useSearchParams } from 'next/navigation';
4
5  export default function SearchBar() {
6    const searchParams = useSearchParams();
7
8    const search = searchParams.get('search');
9
10   // URL -> `/dashboard?search=my-project`
11   // `search` -> 'my-project'
12   return <>Search: {search}</>;
13 }
```

Parameters

```
const searchParams = useSearchParams();
```

useSearchParams does not take any parameters.

Returns

`useSearchParams` returns a **read-only** version of the [URLSearchParams](#) interface, which includes utility methods for reading the URL's query string:

- [URLSearchParams.get\(\)](#): Returns the first value associated with the search parameter. For example:

URL

`searchParams.get("a")`

/dashboard?a=1

'1'

/dashboard?a=

' '

/dashboard?b=3

`null`

/dashboard?a=1&a=2

'1' - use [getAll\(\)](#) to get all values

- [URLSearchParams.has\(\)](#): Returns a boolean value indicating if the given parameter exists. For example:

URL

`searchParams.has("a")`

/dashboard?a=1

true

/dashboard?b=3

false

- Learn more about other **read-only** methods of [URLSearchParams](#), including the [getAll\(\)](#), [keys\(\)](#), [values\(\)](#), [entries\(\)](#), [forEach\(\)](#), and [toString\(\)](#).

Good to know:

- `useSearchParams` is a [Client Component](#) hook and is **not supported** in [Server Components](#) to prevent stale values during [partial rendering](#).
- If an application includes the `/pages` directory, `useSearchParams` will return `ReadonlyURLSearchParams | null`. The `null` value is for compatibility during migration since search params cannot be known during pre-rendering of a page that doesn't use `getServerSideProps`.

Behavior

Static Rendering

If a route is [statically rendered](#), calling `useSearchParams()` will cause the tree up to the closest [Suspense boundary](#) to be client-side rendered.

This allows a part of the page to be statically rendered while the dynamic part that uses `searchParams` is client-side rendered.

You can reduce the portion of the route that is client-side rendered by wrapping the component that uses `useSearchParams` in a `Suspense` boundary. For example:

ts app/dashboard/search-bar.tsx

```
1 'use client';
2
3 import { useSearchParams } from 'next/navigation';
4
5 export default function SearchBar() {
6   const searchParams = useSearchParams();
7
8   const search = searchParams.get('search');
9
10  // This will not be logged on the server when using static rendering
11  console.log(search);
12
13  return <>Search: {search}</>;
14 }
```

ts app/dashboard/page.tsx

```
1 import { Suspense } from 'react';
2 import SearchBar from './search-bar';
3
4 // This component passed as a fallback to the Suspense boundary
5 // will be rendered in place of the search bar in the initial HTML.
6 // When the value is available during React hydration the fallback
7 // will be replaced with the `<SearchBar>` component.
8 function SearchBarFallback() {
9   return <>placeholder</>;
10 }
11
12 export default function Page() {
13   return (
14     <>
15       <nav>
16         <Suspense fallback={<SearchBarFallback />}>
17           <SearchBar />
18         </Suspense>
19       </nav>
20       <h1>Dashboard</h1>
21     </>
22   );
23 }
```

Dynamic Rendering

If a route is [dynamically rendered](#), `useSearchParams` will be available on the server during the initial server render of the Client Component.

Note: Setting the `dynamic` route segment config option to `force-dynamic` can be used to force dynamic rendering.

For example:

TS app/dashboard/search-bar.tsx

```
1 'use client';
2
3 import { useSearchParams } from 'next/navigation';
4
5 export default function SearchBar() {
6   const searchParams = useSearchParams();
7
8   const search = searchParams.get('search');
9
10  // This will be logged on the server during the initial render
11  // and on the client on subsequent navigations.
12  console.log(search);
13
14  return <>Search: {search}</>;
15 }
```

TS app/dashboard/page.tsx

```
1 import SearchBar from './search-bar';
2
3 export const dynamic = 'force-dynamic';
4
5 export default function Page() {
6   return (
7     <>
8       <nav>
9         <SearchBar />
10       </nav>
11       <h1>Dashboard</h1>
12     </>
13   );
14 }
```

Server Components

Pages

To access search params in [Pages](#) (Server Components), use the `searchParams` prop.

Layouts

Unlike Pages, [Layouts](#) (Server Components) **do not** receive the `searchParams` prop. This is because a shared layout is [not re-rendered during navigation](#) which could lead to stale `searchParams` between navigations. View [detailed explanation](#).

Instead, use the Page `searchParams` prop or the `useSearchParams` hook in a Client Component, which is re-rendered on the client with the latest `searchParams`.

Examples

Updating `searchParams`

You can use `useRouter` or `Link` to set new `searchParams`. After a navigation is performed, the current `page.js` will receive an updated `searchParams` prop.

app/example-client-component.tsx



```
1 export default function ExampleClientComponent() {
2   const router = useRouter();
3   const pathname = usePathname();
4   const searchParams = useSearchParams()!;
5
6   // Get a new searchParams string by merging the current
7   // searchParams with a provided key/value pair
8   const createQueryString = useCallback(
9     (name: string, value: string) => {
10       const params = new URLSearchParams(searchParams);
11       params.set(name, value);
12
13       return params.toString();
14     },
15     [searchParams],
16   );
17
18   return (
19     <>
20       <p>Sort By</p>
21
22       /* using useRouter */
23       <button
24         onClick={() => {
25           // <pathname>?sort=asc
26           router.push(pathname + '?' + createQueryString('sort', 'asc'));
27         }}
28     >
```

```
28 >
29     ASC
30 </button>
31
32     /* using <Link> */
33     <Link
34         href={
35             // <pathname>?sort=desc
36             pathname + '?' + createQueryString('sort', 'desc')
37         }
38     >
39         DESC
40     </Link>
41     </>
42 );
43 }
```

> Menu

App Router > ... > Functions > useSelectedLayoutSegment

useSelectedLayoutSegment

`useSelectedLayoutSegment` is a **Client Component** hook that lets you read the active route segment **one level below** the Layout it is called from.

It is useful for navigation UI, such as tabs inside a parent layout that change style depending on the active child segment.

app/example-client-component.tsx

```
1  'use client';
2
3  import { useSelectedLayoutSegment } from 'next/navigation';
4
5  export default function ExampleClientComponent() {
6    const segment = useSelectedLayoutSegment();
7
8    return <>Active segment: {segment}</>;
9  }
```

Good to know:

- Since `useSelectedLayoutSegment` is a **Client Component** hook, and Layouts are **Server Components** by default, `useSelectedLayoutSegment` is usually called via a Client Component that is imported into a Layout.
- `useSelectedLayoutSegment` only returns the segment one level down. To return all active segments, see `useSelectedLayoutSegments`

Parameters

```
const segment = useSelectedLayoutSegment();
```

`useSelectedLayoutSegment` does not take any parameters.

Returns

`useSelectedLayoutSegment` returns a string of the active segment or `null` if one doesn't exist.

For example, given the Layouts and URLs below, the returned segment would be:

Layout	Visited URL	Returned Segment
app/layout.js	/	<code>null</code>
app/layout.js	/dashboard	'dashboard'
app/dashboard/layout.js	/dashboard	<code>null</code>
app/dashboard/layout.js	/dashboard/settings	'settings'
app/dashboard/layout.js	/dashboard/analytics	'analytics'
app/dashboard/layout.js	/dashboard/analytics/monthly	'analytics'

Examples

Creating an active link component

You can use `useSelectedLayoutSegment` to create an active link component that changes style depending on the active segment. For example, a featured posts list in the sidebar of a blog:

TS app/blog/blog-nav-link.tsx

```
1  'use client';
2
3  import Link from 'next/link';
4  import { useSelectedLayoutSegment } from 'next/navigation';
5
6  // This *client* component will be imported into a blog layout
7  export default function BlogNavLink({
8    slug,
9    children,
10  }): {
11    slug: string;
12    children: React.ReactNode;
13  }) {
14    // Navigating to `/blog/hello-world` will return 'hello-world'
15    // for the selected layout segment
```

```
16  const segment = useSelectedLayoutSegment();
17  const isActive = slug === segment;
18
19  return (
20    <Link
21      href={`/blog/${slug}`}
22      // Change style depending on whether the link is active
23      style={{ fontWeight: isActive ? 'bold' : 'normal' }}
24    >
25      {children}
26    </Link>
27  );
28 }
```

app/blog/layout.tsx

```
1 // Import the Client Component into a parent Layout (Server Component)
2 import { BlogNavLink } from './blog-nav-link';
3 import getFeaturedPosts from './get-featured-posts';
4
5 export default async function Layout({
6   children,
7 }: {
8   children: React.ReactNode;
9 }) {
10   const featuredPosts = await getFeaturedPosts();
11   return (
12     <div>
13       {featuredPosts.map((post) => (
14         <div key={post.id}>
15           <BlogNavLink slug={post.slug}>{post.title}</BlogNavLink>
16         </div>
17       )));
18       <div>{children}</div>
19     </div>
20   );
21 }
```

> Menu

App Router > ... > Functions > useSelectedLayoutSegments

useSelectedLayoutSegments

`useSelectedLayoutSegments` is a **Client Component** hook that lets you read the active route segments **below** the Layout it is called from.

It is useful for creating UI in parent Layouts that need knowledge of active child segments such as breadcrumbs.

TS app/example-client-component.tsx

```
1  'use client';
2
3  import { useSelectedLayoutSegments } from 'next/navigation';
4
5  export default function ExampleClientComponent() {
6    const segments = useSelectedLayoutSegments();
7
8    return (
9      <ul>
10        {segments.map((segment, index) => (
11          <li key={index}>{segment}</li>
12        ))}
13      </ul>
14    );
15  }
```

Good to know:

- Since `useSelectedLayoutSegments` is a **Client Component** hook, and Layouts are **Server Components** by default, `useSelectedLayoutSegments` is usually called via a Client Component that is imported into a Layout.
- The returned segments include **Route Groups**, which you might not want to be included in your UI. You can use the `filter()` array method to remove items that start with a bracket.

Parameters

```
const segments = useSelectedLayoutSegments();
```

`useSelectedLayoutSegments` does not take any parameters.

Returns

`useSelectedLayoutSegments` returns an array of strings containing the active segments one level down from the layout the hook was called from. Or an empty array if none exist.

For example, given the Layouts and URLs below, the returned segments would be:

Layout	Visited URL	Returned Segments
app/layout.js	/	[]
app/layout.js	/dashboard	['dashboard']
app/layout.js	/dashboard/settings	['dashboard', 'settings']
app/dashboard/layout.js	/dashboard	[]
app/dashboard/layout.js	/dashboard/settings	['settings']

> Menu

App 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

App 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:

appDir

Enable the App Router to use layouts, streaming, and more.

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

[incrementalCacheHandlerPath](#)

Configure the Next.js cache used for storing and revalidating data.

[mdxRs](#)

Use the new Rust compiler to compile MDX files in the App Router.

[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.

[serverComponentsExternalPackages](#)

Opt-out specific dependencies from the Server Components bundling and use native Node.js `require`.

[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](#)

typedRoutes

Enable experimental support for statically typed links.

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

App Router > ... > next.config.js Options > appDir

appDir

Note: This option is **no longer** needed as of Next.js 13.4. The App Router is now stable.

The App Router ([app](#) directory) enables support for [layouts](#), [Server Components](#), [streaming](#), and [colocated data fetching](#).

Using the [app](#) directory will automatically enable [React Strict Mode](#). Learn how to [incrementally adopt app](#).

> Menu

App 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

App 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

App 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

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

devIndicators

When you edit your code, and Next.js is compiling the application, a compilation indicator appears in the bottom right corner of the page.

Note: This indicator is only present in development mode and will not appear when building and running the app in production mode.

In some cases this indicator can be misplaced on your page, for example, when conflicting with a chat launcher. To change its position, open `next.config.js` and set the `buildActivityPosition` in the `devIndicators` object to `bottom-right` (default), `bottom-left`, `top-right` or `top-left`:

JS `next.config.js`

```
1 module.exports = {  
2   devIndicators: {  
3     buildActivityPosition: 'bottom-right',  
4   },  
5 };
```

In some cases this indicator might not be useful for you. To remove it, open `next.config.js` and disable the `buildActivity` config in `devIndicators` object:

JS `next.config.js`

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


> Menu

App 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

App 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

App 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

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

exportPathMap (Deprecated)

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

App 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

App 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

App 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

App 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

App 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   url.searchParams.set('fm', 'webp');
5   url.searchParams.set('w', width.toString());
6   url.searchParams.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

App Router > ... > next.config.js Options > incrementalCacheHandlerPath

incrementalCacheHandlerPath

In Next.js, the [default cache handler](#) uses the filesystem cache. This requires no configuration, however, you can customize the cache handler by using the `incrementalCacheHandlerPath` field in `next.config.js`.

JS next.config.js

```
1 module.exports = {
2   incrementalCacheHandlerPath: './cache-handler.js',
3 };
```

Here's an example of a custom cache handler:

JS cache-handler.js

```
1 const cache = new Map();
2
3 module.exports = class CacheHandler {
4   constructor(options) {
5     this.options = options;
6     this.cache = {};
7   }
8
9   async get(key) {
10     return cache.get(key);
11   }
12
13   async set(key, data) {
14     cache.set(key, {
15       value: data,
16       lastModified: Date.now(),
17     });
18   }
19 };
```

API Reference

The cache handler can implement the following methods: `get`, `set`, and `revalidateTag`.

get()

Parameter	Type	Description
<code>key</code>	<code>string</code>	The key to the cached value.

Returns the cached value or `null` if not found.

set()

Parameter	Type	Description
<code>key</code>	<code>string</code>	The key to store the data under.
<code>data</code>	Data or <code>null</code>	The data to be cached.

Returns `Promise<void>`.

revalidateTag()

Parameter	Type	Description
<code>tag</code>	<code>string</code>	The cache tag to revalidate.

Returns `Promise<void>`. Learn more about [revalidating data](#) or the `revalidateTag()` function.

> Menu

App Router > ... > next.config.js Options > mdxRs

mdxRs

For use with `@next/mdx`. Compile MDX files using the new Rust compiler.

JS next.config.js

```
1 const withMDX = require('@next/mdx')();
2
3 /** @type {import('next').NextConfig} */
4 const nextConfig = {
5   pageExtensions: ['ts', 'tsx', 'mdx'],
6   experimental: {
7     mdxRs: true,
8   },
9 };
10
11 module.exports = withMDX(nextConfig);
```


> Menu

App 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

App 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

App Router > ... > next.config.js Options > pageExtensions

pageExtensions

By default, Next.js accepts files with the following extensions: `.tsx`, `.ts`, `.jsx`, `.js`. This can be modified to allow other extensions like markdown (`.md`, `.mdx`).

JS next.config.js

```
1 const withMDX = require('@next/mdx')();
2
3 /** @type {import('next').NextConfig} */
4 const nextConfig = {
5   pageExtensions: ['ts', 'tsx', 'mdx'],
6   experimental: {
7     mdxRs: true,
8   },
9 };
10
11 module.exports = withMDX(nextConfig);
```

For custom advanced configuration of Next.js, you can create a `next.config.js` or `next.config.mjs` file in the root of your project directory (next to `package.json`).

`next.config.js` is a regular Node.js module, not a JSON file. It gets used by the Next.js server and build phases, and it's not included in the browser build.

Take a look at the following `next.config.js` example:

JS next.config.js

```
1 /**
2  * @type {import('next').NextConfig}
3 */
4 const nextConfig = {
5   /* config options here */
6 };
7
8 module.exports = nextConfig;
```

If you need [ECMAScript modules ↗](#), you can use `next.config.mjs`:

JS next.config.mjs



```
1  /**
2   * @type {import('next').NextConfig}
3   */
4  const nextConfig = {
5    /* config options here */
6  };
7
8  export default nextConfig;
```

You can also use a function:

JS next.config.mjs



```
1  module.exports = (phase, { defaultConfig }) => {
2    /**
3     * @type {import('next').NextConfig}
4     */
5    const nextConfig = {
6      /* config options here */
7    };
8    return nextConfig;
9  };
```

Since Next.js 12.1.0, you can use an `async` function:

JS next.config.js



```
1  module.exports = async (phase, { defaultConfig }) => {
2    /**
3     * @type {import('next').NextConfig}
4     */
5    const nextConfig = {
6      /* config options here */
7    };
8    return nextConfig;
9  };
```

`phase` is the current context in which the configuration is loaded. You can see the [available phases ↗](#).

Phases can be imported from `next/constants`:

```
1  const { PHASE_DEVELOPMENT_SERVER } = require('next/constants');
```

```
3 module.exports = (phase, { defaultConfig }) => {
4   if (phase === PHASE_DEVELOPMENT_SERVER) {
5     return {
6       /* development only config options here */
7     };
8   }
9
10  return {
11    /* config options for all phases except development here */
12  };
13};
```

The commented lines are the place where you can put the configs allowed by `next.config.js`, which are [defined in this file ↗](#).

However, none of the configs are required, and it's not necessary to understand what each config does. Instead, search for the features you need to enable or modify in this section and they will show you what to do.

Avoid using new JavaScript features not available in your target Node.js version. `next.config.js` will not be parsed by Webpack, Babel or TypeScript.

> Menu

App 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

App 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

App 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

App 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

App 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

App 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

App Router > ... > next.config.js Options > serverComponentsExternalPackages

serverComponentsExternalPackages

Dependencies used inside [Server Components](#) and [Route Handlers](#) will automatically be bundled by Next.js.

If a dependency is using Node.js specific features, you can choose to opt-out specific dependencies from the Server Components bundling and use native Node.js `require`.

JS next.config.js



```
1  /** @type {import('next').NextConfig} */
2  const nextConfig = {
3    experimental: {
4      serverComponentsExternalPackages: ['@acme/ui'],
5    },
6  };
7
8 module.exports = nextConfig;
```

Next.js includes a [short list of popular packages](#) ↗ that currently are working on compatibility and automatically opt-ed out:

- `@prisma/client`
- `@sentry/nextjs`
- `@sentry/node`
- `autoprefixer`
- `aws-crt`
- `bcrypt`
- `cypress`
- `eslint`
- `express`
- `firebase-admin`

- `jest`
- `lodash`
- `mongodb`
- `next-mdx-remote`
- `next-seo`
- `postcss`
- `prettier`
- `prisma`
- `rimraf`
- `sharp`
- `shiki`
- `sqlite3`
- `tailwindcss`
- `ts-node`
- `typescript`
- `vscode-oniguruma`
- `webpack`

> Menu

App 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

App 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

App 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

App Router > ... > next.config.js Options > typedRoutes

typedRoutes (experimental)

Experimental support for [statically typed links](#). This feature requires using the App Router as well as TypeScript in your project.

JS next.config.js



```
1  /** @type {import('next').NextConfig} */
2  const nextConfig = {
3    experimental: {
4      typedRoutes: true,
5    },
6  };
7
8 module.exports = nextConfig;
```


> Menu

App 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

App 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

App 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

App 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

App 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.

Rendering

Learn the differences between Next.js rendering environments, strategies, and runtimes.

Data Fetching

Learn the fundamentals of data fetching with React and 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

App 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 provides an integrated ESLint experience by default. These conformance rules help you use Next.js in an optimal way.

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.

MDX

Learn how to configure MDX to write JSX in your markdown files.

[src Directory](#)

Save pages under the `src` directory as an alternative to the root `pages` directory.

Draft Mode

Next.js has draft mode to toggle between static and dynamic pages. You can learn how it works with App Router here.

> Menu

App 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

App Router > ... > Configuring > Draft Mode

Draft Mode

Static rendering 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 switch to **dynamic rendering** 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 Route Handler

First, create a **Route Handler**. It can have any name - e.g. `app/api/draft/route.ts`

Then, import `draftMode` from `next/headers` and call the `enable()` method.

TS app/api/draft/route.ts

```
1 // route handler enabling draft mode
2 import { draftMode } from 'next/headers';
3
4 export async function GET(request: Request) {
5   draftMode().enable();
6   return new Response('Draft mode is enabled');
7 }
```

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 visiting `/api/draft` and looking at your browser's developer tools. Notice the `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 Route Handler *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 Route Handler is located at `app/api/draft/route.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 Route Handler:

- Check that the secret matches and that the `slug` parameter exists (if not, the request should fail).
- Call `draftMode.enable()` to set the cookie.
- Then redirect the browser to the path specified by `slug`.

ts app/api/draft/route.ts



```
1 // route handler with secret and slug
2 import { draftMode } from 'next/headers';
3 import { redirect } from 'next/navigation';
4
5 export async function GET(request: Request) {
6   // Parse query string parameters
7   const { searchParams } = new URL(request.url);
8   const secret = searchParams.get('secret');
9   const slug = searchParams.get('slug');
```

```
10
11 // Check the secret and next parameters
12 // This secret should only be known to this route handler and the CMS
13 if (secret !== 'MY_SECRET_TOKEN' || !slug) {
14   return new Response('Invalid token', { status: 401 });
15 }
16
17 // Fetch the headless CMS to check if the provided `slug` exists
18 // getPostBySlug would implement the required fetching logic to the headless CMS
19 const post = await getPostBySlug(slug);
20
21 // If the slug doesn't exist prevent draft mode from being enabled
22 if (!post) {
23   return new Response('Invalid slug', { status: 401 });
24 }
25
26 // Enable Draft Mode by setting the cookie
27 draftMode().enable();
28
29 // Redirect to the path from the fetched post
30 // We don't redirect to searchParams.slug as that might lead to open redirect vulnerability
31 redirect(post.slug);
32 }
```

If it succeeds, then the browser will be redirected to the path you want to view with the draft mode cookie.

Step 2: Update page

The next step is to update your page to check the value of `draftMode().isEnabled`.

If you request a page which has the cookie set, then data will be fetched at **request time** (instead of at build time).

Furthermore, the value of `isEnabled` will be `true`.

app/page.tsx

```
1 // page that fetches data
2 import { draftMode } from 'next/headers';
3
4 async function getData() {
5   const { isEnabled } = draftMode();
6
7   const url = isEnabled
8     ? 'https://draft.example.com'
9     : 'https://production.example.com';
10
```

```
11  const res = await fetch(url);
12
13  return res.json();
14 }
15
16 export default async function Page() {
17  const { title, desc } = await getData();
18
19  return (
20    <main>
21      <h1>{title}</h1>
22      <p>{desc}</p>
23    </main>
24  );
25 }
```

That's it! If you access the draft Route Handler (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 a Route Handler that calls `draftMode().disable()`:

ts app/api/disable-draft/route.ts



```
1 import { draftMode } from 'next/headers';
2
3 export async function GET(request: Request) {
4   draftMode().disable();
5   return new Response('Draft mode is disabled');
6 }
```

Then, send a request to `/api/disable-draft` to invoke the Route Handler. If calling this route using `next/link`, you must pass `prefetch={false}` to prevent accidentally deleting the cookie on prefetch.

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

App 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 Route Handlers.

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.

```
js 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('abcdefghijklm')`.
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);
7 }
```

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

App 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

App 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 `@next/mdx` package:

>_ Terminal

```
npm install @next/mdx @mdx-js/loader @mdx-js/react @types/mdx
```

Create `mdx-components.tsx` in the root of your application (the parent folder of `app`):

ts mdx-components.tsx

```
1 import type { MDXComponents } from 'mdx/types';
2
3 // This file allows you to provide custom React components
4 // to be used in MDX files. You can import and use any
5 // React component you want, including components from
6 // other libraries.
7
8 // This file is required to use MDX in `app` directory.
9 export function useMDXComponents(components: MDXComponents): MDXComponents {
10   return {
11     // Allows customizing built-in components, e.g. to add styling.
12     // h1: ({ children }) => <h1 style={{ fontSize: "100px" }}>{children}</h1>,
13     ...components,
14   };
15 }
```

Update `next.config.js` to use `mdxRs`:

js next.config.js

```
1 /** @type {import('next').NextConfig} */
2 const nextConfig = {
3   experimental: {
4     mdxRs: true,
5   },
6 };
7
8 const withMDX = require('@next/mdx')();
9 module.exports = withMDX(nextConfig);
```

Add a new file with MDX content to your `app` directory:



```
1 Hello, Next.js!
2
3 You can import and use React components in MDX files.
```

Import the MDX file inside a `page` to display the content:



```
1 import HelloWorld from './hello.mdx';
2
3 export default function Page() {
4   return <HelloWorld />;
5 }
```

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).



```
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 share a layout around MDX content, you can use the [built-in layouts support](#) with the App Router.

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.

JS next.config.js

```
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

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`.

 `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`:

`JS` next.config.js

```
1 module.exports = withMDX({
2   experimental: {
3     mdxRs: true,
4   },
5 });
```

Helpful Links

- [MDX ↗](#)
- [@next/mdx ↗](#)
- [remark ↗](#)
- [rehype ↗](#)

> Menu

App 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`.

Next Steps

App Router > ... > Routing

Project Organization

Learn how to organize your Next.js project and colocate files.

> Menu

App 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.

As well as a [TypeScript Plugin](#) for your editor.

Watch: Learn about the built-in TypeScript plugin → [YouTube \(3 minutes\)](#) ↗

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.

TypeScript Plugin

Next.js includes a custom TypeScript plugin and type checker, which VSCode and other code editors can use for advanced type-checking and auto-completion.

The first time you run `next dev` with a TypeScript file open, you will receive a prompt to enable the plugin.

If you miss the prompt, you can enable the plugin manually by:

1. Opening the command palette (`Ctrl/⌘ + Shift + P`)
2. Searching for "TypeScript: Select TypeScript Version"
3. Selecting "Use Workspace Version"

Now, when editing files, the custom plugin will be enabled. When running `next build`, the custom type checker will be used. Further, we automatically create a VSCode settings file for you to automate this process.

Plugin Features

The TypeScript plugin can help with:

- Warning if the invalid values for [segment config options](#) are passed.
- Showing available options and in-context documentation.
- Ensuring the `use client` directive is used correctly.
- Ensuring client hooks (like `useState`) are only used in Client Components.

Note: More features will be added in the future.

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 ↗](#).

Statically Typed Links

Next.js can statically type links to prevent typos and other errors when using `next/link`, improving type safety when navigating between pages.

To opt-into this feature, `experimental.typedRoutes` need to be enabled and the project needs to be using TypeScript.

`js` next.config.js

```
1  /** @type {import('next').NextConfig} */
2  const nextConfig = {
3    experimental: {
4      typedRoutes: true,
5    },
6  };
7
8 module.exports = nextConfig;
```



Next.js will generate a link definition in `.next/types` that contains information about all existing routes in your application, which TypeScript can then use to provide feedback in your editor about invalid links.

Currently, experimental support includes any string literal, including dynamic segments. For non-literal strings, you currently need to manually cast the `href` with `as Route`:

```
1 import type { Route } from 'next';
2 import Link from 'next/link'
3
4 // No TypeScript errors if href is a valid route
5 <Link href="/about" />
6 <Link href="/blog/nextjs" />
7 <Link href={`/blog/${slug}`} />
8 <Link href={'/blog' + slug} as Route> />
9
10 // TypeScript errors if href is not a valid route
11 <Link href="/aboot" />
```

To accept `href` in a custom component wrapping `next/link`, use a generic:

```
1 import type { Route } from 'next';
2 import Link from 'next/link';
3
4 function Card<T extends string>({ href }: { href: Route<T> | URL })
5   return (
6     <Link href={href}>
7       <div>My Card</div>
8     </Link>
9   );
10 }
```

How does it work?

When running `next dev` or `next build`, Next.js generates a hidden `.d.ts` file inside `.next` that contains information about all existing routes in your application (all valid routes as the `href` type of `Link`). This `.d.ts` file is included in `tsconfig.json` and the TypeScript compiler will check that `.d.ts` and provide feedback in your editor about invalid links.

End-to-End Type Safety

Next.js 13 has **enhanced type safety**. This includes:

1. **No serialization of data between fetching function and page:** You can `fetch` directly in components, layouts, and pages on the server. This data *does not* need to be serialized (converted to a string) to be passed to the client side for consumption in React. Instead, since `app` uses Server Components by default,

we can use values like `Date`, `Map`, `Set`, and more without any extra steps. Previously, you needed to manually type the boundary between server and client with Next.js-specific types.

2. Streamlined data flow between components: With the removal of `_app` in favor of root layouts, it is now easier to visualize the data flow between components and pages. Previous, data flowing between individual `pages` and `_app` were difficult to type and could introduce confusing bugs. With [colocated data fetching](#) in Next.js 13, this is no longer an issue.

[Data Fetching in Next.js](#) now provides as close to end-to-end type safety as possible without being prescriptive about your database or content provider selection.

We're able to type the response data as you would expect with normal TypeScript. For example:

TS app/page.tsx

```
1  async function getData() {
2    const res = await fetch('https://api.example.com/...');
3    // The return value is *not* serialized
4    // You can return Date, Map, Set, etc.
5    return res.json();
6  }
7
8  export default async function Page() {
9    const name = await getData();
10   return '...';
11 }
```

For *complete* end-to-end type safety, this also requires your database or content provider to support TypeScript. This could be through using an [ORM ↗](#) or type-safe query builder.

Async Server Component TypeScript Error

To use an `async` Server Component with TypeScript, ensure you are using TypeScript `5.2` or higher and `@types/react` `18.2.8` or higher.

If you are using an older version of TypeScript, you may see a

`'Promise<Element>' is not a valid JSX element` type error. Updating to the latest version of TypeScript and `@types/react` should resolve this issue.

Passing Data Between Server & Client Components

When passing data between a Server and Client Component through props, the data is still serialized (converted to a string) for use in the browser. However, it does not need a special type. It's typed the same as passing any other props between components.

Further, there is less code to be serialized, as un-rendered data does not cross between the server and client (it remains on the server). This is only now possible through support for Server Components.

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

App Router > Building Your Application > Data Fetching

Data Fetching

The Next.js App Router introduces a new, simplified data fetching system built on React and the Web platform. This page will go through the fundamental concepts and patterns to help you manage your data's lifecycle.

Here's a quick overview of the recommendations on this page:

1. [Fetch data on the server](#) using Server Components.
2. [Fetch data in parallel](#) to minimize waterfalls and reduce loading times.
3. For Layouts and Pages, [fetch data where it's used](#). Next.js will automatically dedupe requests in a tree.
4. Use [Loading UI, Streaming and Suspense](#) to progressively render a page and show a result to the user while the rest of the content loads.

The `fetch()` API

The new data fetching system is built on top of the native [fetch\(\) Web API ↗](#) and makes use of `async` and `await` in Server Components.

- React extends `fetch` to provide [automatic request deduping](#).
- Next.js extends the `fetch` options object to allow each request to set its own [caching and revalidating](#) rules.

[Learn how to use `fetch` in Next.js.](#)

Fetching Data on the Server

Whenever possible, we recommend fetching data in [Server Components](#). Server Components **always fetch data on the server**. This allows you to:

- Have direct access to backend data resources (e.g. databases).
- Keep your application more secure by preventing sensitive information, such as access tokens and API keys, from being exposed to the client.
- Fetch data and render in the same environment. This reduces both the back-and-forth communication between client and server, as well as the work on the main thread on the client.
- Perform multiple data fetches with single round-trip instead of multiple individual requests on the client.
- Reduce client-server [waterfalls](#).
- Depending on your region, data fetching can also happen closer to your data source, reducing latency and improving performance.

Good to know: It's still possible to fetch data client-side. We recommend using a third-party library such as [SWR](#) or [React Query](#) with Client Components. In the future, it'll also be possible to fetch data in Client Components using React's [use\(\)](#) hook.

Fetching Data at the Component Level

In the App Router, you can fetch data inside [layouts](#), [pages](#), and components. Data fetching is also compatible with [Streaming and Suspense](#).

Good to know: For layouts, it's not possible to pass data between a parent layout and its `children` components. We recommend **fetching data directly inside the layout that needs it**, even if you're requesting the same data multiple times in a route. Behind the scenes, React and Next.js will [cache and dedupe](#) requests to avoid the same data being fetched more than once.

Parallel and Sequential Data Fetching

When fetching data inside components, you need to be aware of two data fetching patterns: Parallel and Sequential.

- With **parallel data fetching**, requests in a route are eagerly initiated and will load data at the same time. This reduces client-server waterfalls and the total time it takes to load data.
- With **sequential data fetching**, requests in a route are dependent on each other and create waterfalls. There may be cases where you want this pattern because one fetch depends on the result of the other, or you want a condition to be satisfied before the next fetch to save resources. However, this behavior can also be unintentional and lead to longer loading times.

[Learn how to implement parallel and sequential data fetching.](#)

Automatic `fetch()` Request Deduping

If you need to fetch the same data (e.g. current user) in multiple components in a tree, Next.js will automatically cache `fetch` requests (`GET`) that have the same input in a temporary cache. This optimization prevents the same data from being fetched more than once during a rendering pass.

- On the server, the cache lasts the lifetime of a server request until the rendering process completes.
 - This optimization applies to `fetch` requests made in Layouts, Pages, Server Components, `generateMetadata` and `generateStaticParams`.
 - This optimization also applies during [static generation](#).
- On the client, the cache lasts the duration of a session (which could include multiple client-side re-renders) before a full page reload.

Good to know:

- `POST` requests are not automatically deduplicated. [Learn more about caching](#).
- If you're unable to use `fetch`, React provides a `cache` function to allow you to manually cache data for the duration of the request.

Static and Dynamic Data Fetching

There are two types of data: **Static** and **Dynamic**.

- **Static Data** is data that doesn't change often. For example, a blog post.
- **Dynamic Data** is data that changes often or can be specific to users. For example, a shopping cart list.

By default, Next.js automatically does static fetches. This means that the data will be fetched at build time, cached, and reused on each request. As a developer, you have control over how the static data is [cached](#) and [revalidated](#).

There are two benefits to using static data:

1. It reduces the load on your database by minimizing the number of requests made.
2. The data is automatically cached for improved loading performance.

However, if your data is personalized to the user or you want to always fetch the latest data, you can mark requests as *dynamic* and fetch data on each request without caching.

[Learn how to do Static and Dynamic data fetching.](#)

Caching Data

Caching is the process of storing data in a location (e.g. [Content Delivery Network ↗](#)) so it doesn't need to be re-fetched from the original source on each request.

The **Next.js Cache** is a persistent [HTTP cache ↗](#) that can be globally distributed. This means the cache can scale automatically and be shared across multiple regions depending on your platform (e.g. [Vercel ↗](#)).

Next.js extends the [options object ↗](#) of the `fetch()` function to allow each request on the server to set its own persistent caching behavior. Together with [component-level data fetching](#), this allows you to configure caching within your application code directly where the data is being used.

During server rendering, when Next.js comes across a `fetch`, it will check the cache to see if the data is already available. If it is, it will return the cached data. If not, it will fetch and store data for future requests.

Good to know: If you're unable to use `fetch`, React provides a `cache` function to allow you to manually cache data for the duration of the request.

[Learn more about caching in Next.js.](#)

Revalidating Data

Revalidation is the process of purging the cache and re-fetching the latest data. This is useful when your data changes and you want to ensure your application shows the latest version without having to rebuild your entire application.

Next.js provides two types of revalidation:

- **Background:** Revalidates the data at a specific time interval.
- **On-demand:** Revalidates the data whenever there is an update.

[Learn how to revalidate data.](#)

Streaming and Suspense

Streaming and [Suspense](#) are new React features that allow you to progressively render and incrementally stream rendered units of the UI to the client.

With Server Components and [nested layouts](#), you're able to instantly render parts of the page that do not specifically require data, and show a [loading state](#) for parts of the page that are fetching data. This means the user does not have to wait for the entire page to load before they can start interacting with it.

To learn more about Streaming and Suspense, see the [Loading UI](#) and [Streaming and Suspense](#) pages.

Old Methods

Previous Next.js data fetching methods such as `getServerSideProps`, `getStaticProps`, and `getInitialProps` are **not** supported in the new App Router. However, you can still use them in the [Pages Router](#).

Next Steps

Now that you understand the fundamentals of data fetching in Next.js, you can learn more about managing data in your application:

Fetching

Learn how to fetch data in your Next.js application.

Caching

Learn about caching routes in Next.js.

Revalidating

Learn about revalidating data in Next.js using Incremental Static Regeneration.

Server Actions

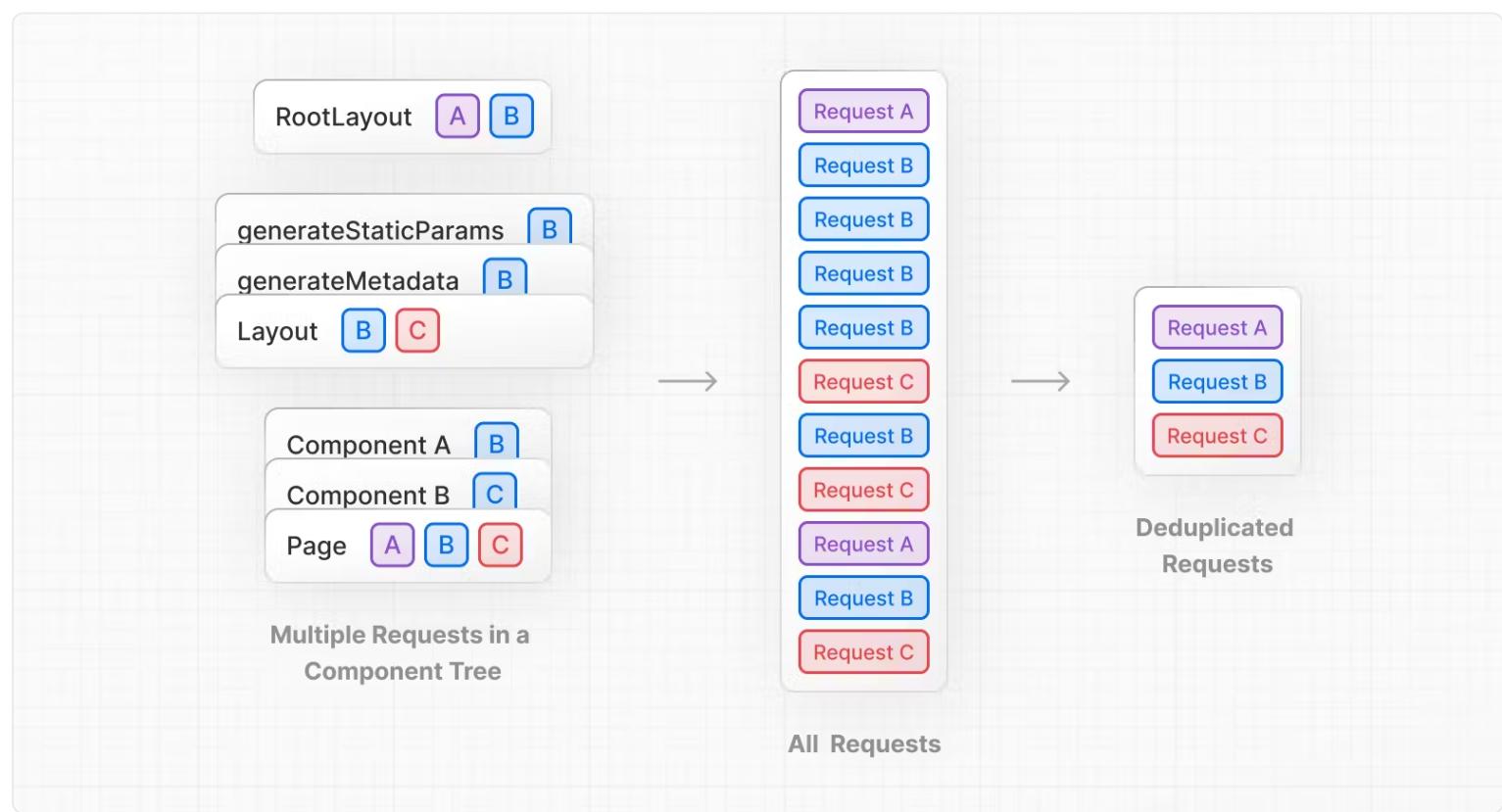
Use Server Actions to mutate data in your Next.js application.

> Menu

App Router > ... > Data Fetching > Caching

Caching Data

Next.js has built-in support for caching data, both on a per-request basis (recommended) or for an entire route segment.



Per-request Caching

fetch()

By default, all `fetch()` requests are cached and deduplicated automatically. This means that if you make the same request twice, the second request will reuse the result from the first request.

```
1 async function getComments() {  
2   const res = await fetch('https://...'); // The result is cached  
3   return res.json();  
4 }  
5  
6 // This function is called twice, but the result is only fetched once  
7 const comments = await getComments(); // cache MISS  
8  
9 // The second call could be anywhere in your application  
10 const comments = await getComments(); // cache HIT
```

Requests are **not** cached if:

- Dynamic methods (`next`/`headers`, `export const POST`, or similar) are used and the fetch is a `POST` request (or uses `Authorization` or `cookie` headers)
- `fetchCache` is configured to skip cache by default
- `revalidate: 0` or `cache: 'no-store'` is configured on individual `fetch`

Requests made using `fetch` can specify a `revalidate` option to control the revalidation frequency of the request.

TS app/page.tsx

```
1 export default async function Page() {  
2   // revalidate this data every 10 seconds at most  
3   const res = await fetch('https://...', { next: { revalidate: 10 } });  
4   const data = res.json();  
5   // ...  
6 }
```

React `cache()`

React allows you to `cache()` ↗ and deduplicate requests, memoizing the result of the wrapped function call. The same function called with the same arguments will reuse a cached value instead of re-running the function.

TS utils/getUser.ts

```
1 import { cache } from 'react';  
2  
3 export const getUser = cache(async (id: string) => {  
4   const user = await db.user.findUnique({ id });  
5   return user;  
6 });
```

```

1 import { getUser } from '@utils/getUser';
2
3 export default async function UserLayout({ params: { id } }) {
4   const user = await getUser(id);
5   // ...
6 }

```

```

1 import { getUser } from '@utils/getUser';
2
3 export default async function Page({
4   params: { id },
5 }: {
6   params: { id: string };
7 }) {
8   const user = await getUser(id);
9   // ...
10 }

```

Although the `getUser()` function is called twice in the example above, only one query will be made to the database. This is because `getUser()` is wrapped in `cache()`, so the second request can reuse the result from the first request.

Good to know:

- `fetch()` caches requests automatically, so you don't need to wrap functions that use `fetch()` with `cache()`. See [automatic request deduping](#) for more information.
- In this new model, we recommend **fetching data directly in the component that needs it**, even if you're requesting the same data in multiple components, rather than passing the data between components as props.
- We recommend using the [server-only package](#) to make sure server data fetching functions are never used on the client.

POST requests and `cache()`

`POST` requests are automatically deduplicated when using `fetch` – unless they are inside of `POST` Route Handler or come after reading `headers()` / `cookies()`. For example, if you are using GraphQL and `POST` requests in the above cases, you can use `cache` to deduplicate requests. The `cache` arguments must be flat and only include primitives. Deep objects won't match for deduplication.

```
1 import { cache } from 'react';
2
3 export const getUser = cache(async (id: string) => {
4   const res = await fetch('...', { method: 'POST', body: '...' });
5   // ...
6});
```

Preload pattern with `cache()`

As a pattern, we suggest optionally exposing a `preload()` export in utilities or components that do data fetching.

components/User.tsx

```
1 import { getUser } from '@utils/getUser';
2
3 export const preload = (id: string) => {
4   // void evaluates the given expression and returns undefined
5   // https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void
6   void getUser(id);
7 };
8 export default async function User({ id }: { id: string }) {
9   const result = await getUser(id);
10  // ...
11 }
```

By calling `preload`, you can eagerly start fetching data you're likely going to need.

app/user/[id]/page.tsx

```
1 import User, { preload } from '@components/User';
2
3 export default async function Page({
4   params: { id },
5 }: {
6   params: { id: string };
7 }) {
8   preload(id); // starting loading the user data now
9   const condition = await fetchCondition();
10  return condition ? <User id={id} /> : null;
11 }
```

Good to know:

- The `preload()` function can have any name. It's a pattern, not an API.
- This pattern is completely optional and something you can use to optimize on a case-by-case basis. This pattern is a further optimization on top of `parallel data fetching`. Now you don't have to pass promises down as props and can

instead rely on the preload pattern.

Combining `cache`, `preload`, and `server-only`

You can combine the `cache` function, the `preload` pattern, and the `server-only` package to create a data fetching utility that can be used throughout your app.

utils/getUser.ts

```
1 import { cache } from 'react';
2 import 'server-only';
3
4 export const preload = (id: string) => {
5   void getUser(id);
6 };
7
8 export const getUser = cache(async (id: string) => {
9   // ...
10});
```

With this approach, you can eagerly fetch data, cache responses, and guarantee that this data fetching [only happens on the server](#).

The `getUser.ts` exports can be used by layouts, pages, or components to give them control over when a user's data is fetched.

Segment-level Caching

Note: We recommend using per-request caching for improved granularity and control over caching.

Segment-level caching allows you to cache and revalidate data used in route segments.

This mechanism allows different segments of a path to control the cache lifetime of the entire route. Each `page.tsx` and `layout.tsx` in the route hierarchy can export a `revalidate` value that sets the revalidation time for the route.

app/page.tsx

```
export const revalidate = 60; // revalidate this segment every 60 seconds
```

Good to know:

- If a page, layout, and fetch request inside components all specify a `revalidate` frequency, the lowest value of the three will be used.
- Advanced: You can set `fetchCache` to `'only-cache'` or `'force-cache'` to ensure that all `fetch` requests opt into caching but the revalidation frequency might still be lowered by individual `fetch` requests. See `fetchCache` for more information.

> Menu

App Router > ... > Data Fetching > Fetching

Data Fetching

The Next.js App Router allows you to fetch data directly in your React components by marking the function as `async` and using `await` for the [Promise ↗](#).

Data fetching is built on top of the [fetch\(\) Web API ↗](#) and React Server Components. When using `fetch()`, requests are [automatically deduped](#) by default.

Next.js extends the `fetch` options object to allow each request to set its own [caching and revalidating](#).

async and await in Server Components

You can use `async` and `await` to fetch data in Server Components.

ts app/page.tsx

```
1  async function getData() {
2    const res = await fetch('https://api.example.com/...');
3    // The return value is *not* serialized
4    // You can return Date, Map, Set, etc.
5
6    // Recommendation: handle errors
7    if (!res.ok) {
8      // This will activate the closest `error.js` Error Boundary
9      throw new Error('Failed to fetch data');
10   }
11
12   return res.json();
13 }
14
15 export default async function Page() {
16   const data = await getData();
17
18   return <main></main>;
19 }
```

Good to know:

To use an `async` Server Component with TypeScript, ensure you are using TypeScript [5.2](#) or higher and `@types/react` [18.2.8](#) or higher.

Server Component Functions

Next.js provides helpful server functions you may need when fetching data in Server Components:

- `cookies()`
- `headers()`

use in Client Components

`use` is a new React function that **accepts a promise** conceptually similar to `await`. `use` **handles the promise** returned by a function in a way that is compatible with components, hooks, and Suspense. Learn more about `use` in the [React RFC ↗](#).

Wrapping `fetch` in `use` is currently **not** recommended in Client Components and may trigger multiple re-renders. For now, if you need to fetch data in a Client Component, we recommend using a third-party library such as [SWR ↗](#) or [React Query ↗](#).

Note: We'll be adding more examples once `fetch` and `use` work in Client Components.

Static Data Fetching

By default, `fetch` will automatically fetch and **cache data** indefinitely.

```
fetch('https://...'); // cache: 'force-cache' is the default
```

Revalidating Data

To revalidate **cached data** at a timed interval, you can use the `next.revalidate` option in `fetch()` to set the `cache` lifetime of a resource (in seconds).

```
fetch('https://...', { next: { revalidate: 10 } });
```

See [Revalidating Data](#) for more information.

Good to know:

Caching at the fetch level with `revalidate` or `cache: 'force-cache'` stores the data across requests in a shared cache. You should avoid using it for user-specific data (i.e. requests that derive data from `cookies()` or `headers()`)

Dynamic Data Fetching

To fetch fresh data on every `fetch` request, use the `cache: 'no-store'` option.

```
fetch('https://...', { cache: 'no-store' });
```

Data Fetching Patterns

Parallel Data Fetching

To minimize client-server waterfalls, we recommend this pattern to fetch data in parallel:

```
ts app/artist/[username]/page.tsx
```

```
1 import Albums from './albums';
2
3 async function getArtist(username: string) {
4   const res = await fetch(`https://api.example.com/artist/${username}`);
5   return res.json();
6 }
7
8 async function getArtistAlbums(username: string) {
9   const res = await fetch(`https://api.example.com/artist/${username}/albums`);
10  return res.json();
11 }
12
13 export default async function Page({
14   params: { username },
15 }: {
16   params: { username: string };
17 }) {
```

```

18 // Initiate both requests in parallel
19 const artistData = getArtist(username);
20 const albumsData = getArtistAlbums(username);
21
22 // Wait for the promises to resolve
23 const [artist, albums] = await Promise.all([artistData, albumsData]);
24
25 return (
26   <>
27   <h1>{artist.name}</h1>
28   <Albums list={albums}></Albums>
29   </>
30 );
31 }

```

By starting the fetch prior to calling `await` in the Server Component, each request can eagerly start to fetch requests at the same time. This sets the components up so you can avoid waterfalls.

We can save time by initiating both requests in parallel, however, the user won't see the rendered result until both promises are resolved.

To improve the user experience, you can add a [suspense boundary](#) to break up the rendering work and show part of the result as soon as possible:



```

TS artist/[username]/page.tsx

```

```

1 import { getArtist, getArtistAlbums, type Album } from './api';
2
3 export default async function Page({
4   params: { username },
5 }: {
6   params: { username: string };
7 }) {
8   // Initiate both requests in parallel
9   const artistData = getArtist(username);
10  const albumData = getArtistAlbums(username);
11
12  // Wait for the artist's promise to resolve first
13  const artist = await artistData;
14
15  return (
16    <>
17    <h1>{artist.name}</h1>
18    /* Send the artist information first,
19       and wrap albums in a suspense boundary */
20    <Suspense fallback=<div>Loading...</div>>
21      <Albums promise={albumData} />
22    </Suspense>
23    </>
24  );
25 }

```

```
26
27 // Albums Component
28 async function Albums({ promise }: { promise: Promise<Album[]> }) {
29   // Wait for the albums promise to resolve
30   const albums = await promise;
31
32   return (
33     <ul>
34       {albums.map((album) => (
35         <li key={album.id}>{album.name}</li>
36       ))}
37     </ul>
38   );
39 }
```

Take a look at the [preloading pattern](#) for more information on improving components structure.

Sequential Data Fetching

To fetch data sequentially, you can `fetch` directly inside the component that needs it, or you can `await` the result of `fetch` inside the component that needs it:



```
1 // ...
2
3 async function Playlists({ artistID }: { artistID: string }) {
4   // Wait for the playlists
5   const playlists = await getArtistPlaylists(artistID);
6
7   return (
8     <ul>
9       {playlists.map((playlist) => (
10         <li key={playlist.id}>{playlist.name}</li>
11       ))}
12     </ul>
13   );
14 }
15
16 export default async function Page({
17   params: { username },
18 }: {
19   params: { username: string };
20 }) {
21   // Wait for the artist
22   const artist = await getArtist(username);
23
24   return (
25     <>
26       <h1>{artist.name}</h1>
27       <Suspense fallback=<div>Loading...</div>>
```

```
28      <Playlists artistID={artist.id} />
29    </Suspense>
30  </>
31  );
32 }
```

By fetching data inside the component, each fetch request and nested segment in the route cannot start fetching data and rendering until the previous request or segment has completed.

Blocking Rendering in a Route

By fetching data in a [layout](#), rendering for all route segments beneath it can only start once the data has finished loading.

In the `pages` directory, pages using server-rendering would show the browser loading spinner until `getServerSideProps` had finished, then render the React component for that page. This can be described as "all or nothing" data fetching. Either you had the entire data for your page, or none.

In the `app` directory, you have additional options to explore:

1. First, you can use `loading.js` to show an instant loading state from the server while streaming in the result from your data fetching function.
2. Second, you can move data fetching *lower* in the component tree to only block rendering for the parts of the page that need it. For example, moving data fetching to a specific component rather than fetching it at the root layout.

Whenever possible, it's best to fetch data in the segment that uses it. This also allows you to show a loading state for only the part of the page that is loading, and not the entire page.

Data Fetching without `fetch()`

You might not always have the ability to use and configure `fetch` requests directly if you're using a third-party library such as an ORM or database client.

In cases where you cannot use `fetch` but still want to control the caching or revalidating behavior of a layout or page, you can rely on the [default caching behavior](#) of the segment or use the [segment cache configuration](#).

Default Caching Behavior

Any data fetching libraries that do not use `fetch` directly **will not** affect caching of a route, and will be static or dynamic depending on the route segment.

If the segment is static (default), the output of the request will be cached and revalidated (if configured) alongside the rest of the segment. If the segment is dynamic, the output of the request will *not* be cached and will be re-fetched on every request when the segment is rendered.

Good to know: Dynamic functions like `cookies()` and `headers()` will make the route segment dynamic.

Segment Cache Configuration

As a temporary solution, until the caching behavior of third-party queries can be configured, you can use [segment configuration](#) to customize the cache behavior of the entire segment.

TS app/page.tsx

```
1 import prisma from './lib/prisma';
2
3 export const revalidate = 3600; // revalidate every hour
4
5 async function getPosts() {
6   const posts = await prisma.post.findMany();
7   return posts;
8 }
9
10 export default async function Page() {
11   const posts = await getPosts();
12   // ...
13 }
```

> Menu

App Router > ... > Data Fetching > Revalidating

Revalidating Data

Next.js allows you to update specific static routes **without needing to rebuild your entire site**. Revalidation (also known as [Incremental Static Regeneration](#)) allows you to retain the benefits of static while scaling to millions of pages.

There are two types of revalidation in Next.js:

- **Background:** Revalidates the data at a specific time interval.
- **On-demand:** Revalidates the data based on an event such as an update.

Background Revalidation

To revalidate cached data at a specific interval, you can use the `next.revalidate` option in `fetch()` to set the `cache` lifetime of a resource (in seconds).

```
fetch('https://...', { next: { revalidate: 60 } });
```

If you want to revalidate data that does not use `fetch` (i.e. using an external package or query builder), you can use the [route segment config](#).

TS app/page.tsx

```
export const revalidate = 60; // revalidate this page every 60 seconds
```

In addition to `fetch`, you can also revalidate data using `cache`.

How it works

1. When a request is made to the route that was statically rendered at build time, it will initially show the cached data.
2. Any requests to the route after the initial request and before 60 seconds are also cached and instantaneous.
3. After the 60-second window, the next request will still show the cached (stale) data.
4. Next.js will trigger a regeneration of the data in the background.
5. Once the route generates successfully, Next.js will invalidate the cache and show the updated route. If the background regeneration fails, the old data would still be unaltered.

When a request is made to a route segment that hasn't been generated, Next.js will dynamically render the route on the first request. Future requests will serve the static route segments from the cache.

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. ISR on Vercel [persists the cache globally and handles rollbacks ↗](#).

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 if someone visits the page after the minute has passed.

The Next.js App Router supports revalidating content on-demand based on a route or cache tag. This allows you to manually purge the Next.js cache for specific fetches, making it easier to update your site when:

- Content from your headless CMS is created or updated.
- Ecommerce metadata changes (price, description, category, reviews, etc).

Using On-Demand Revalidation

Data can be revalidated on-demand by path (`revalidatePath`) or by cache tag (`revalidateTag`).

For example, the following `fetch` adds the cache tag `collection`:

app/page.tsx

```
1 export default async function Page() {  
2   const res = await fetch('https://...', { next: { tags: ['collection'] } });
```

```
3   const data = await res.json();
4   // ...
5 }
```

This cached data can then be revalidated on-demand by calling `revalidateTag` in a [Route Handler](#).

app/api/revalidate/route.ts

```
1 import { NextRequest, NextResponse } from 'next/server';
2 import { revalidateTag } from 'next/cache';
3
4 export async function GET(request: NextRequest) {
5   const tag = request.nextUrl.searchParams.get('tag');
6   revalidateTag(tag);
7   return NextResponse.json({ revalidated: true, now: Date.now() });
8 }
```

Error Handling and Revalidation

If an error is thrown while attempting to revalidate data, the last successfully generated data will continue to be served from the cache. On the next subsequent request, Next.js will retry revalidating the data.

> Menu

App Router > ... > Data Fetching > Server Actions

Server Actions

Server Actions are an **alpha** feature in Next.js, built on top of React Actions. They enable server-side data mutations, reduced client-side JavaScript, and progressively enhanced forms. They can be defined inside Server Components and/or called from Client Components:

With Server Components:

 app/add-to-cart.jsx

```
1 import { cookies } from 'next/headers';
2
3 // Server action defined inside a Server Component
4 export default function AddToCart({ productId }) {
5   async function addItem(data) {
6     'use server';
7
8     const cartId = cookies().get('cartId')?.value;
9     await saveToDb({ cartId, data });
10   }
11
12   return (
13     <form action={addItem}>
14       <button type="submit">Add to Cart</button>
15     </form>
16   );
17 }
```

With Client Components:

 app/actions.js

```
1 'use server';
2
3 async function addItem(data) {
4   const cartId = cookies().get('cartId')?.value;
5   await saveToDb({ cartId, data });
6 }
```



```
1 'use client';
2
3 import { addItem } from './actions.js';
4
5 // Server Action being called inside a Client Component
6 export default function AddToCart({ productId }) {
7   return (
8     <form action={addItem}>
9       <button type="submit">Add to Cart</button>
10    </form>
11  );
12}
```

Convention

You can enable Server Actions in your Next.js project by enabling the **experimental serverActions** flag.



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

Creation

Server Actions can be defined in two places:

- Inside the component that uses it (Server Components only)
- In a separate file (Client and Server Components), for reusability. You can define multiple Server Actions in a single file.

With Server Components

Create a Server Action by defining an asynchronous function with the **"use server"** directive at the top of the function body. This function should have **serializable arguments** and a **serializable return value** based on the React Server Components protocol.



```

1  export default function ServerComponent() {
2    async function myAction() {
3      'use server';
4      // ...
5    }
6  }

```

With Client Components

If you're using a Server Action inside a Client Component, create your action in a separate file with the "use server" directive at the top of the file. Then, import the Server Action into your Client Component:



```

1  'use server';
2
3  export async function myAction() {
4    // ...
5  }

```



```

1  | "use client"
2
3  import { myAction } from './actions';
4
5  export default function ClientComponent() {
6    return (
7      <form action={myAction}>
8        <button type="submit">Add to Cart</button>
9      </form>
10    );
11  }

```

Note: When using a top-level "use server" directive, all exports below will be considered Server Actions. You can have multiple Server Actions in a single file.

Invocation

You can invoke Server Actions using the following methods:

- Using `action`: React's `action` prop allows invoking a Server Action on a `<form>` element.

- Using `formAction`: React's `formAction` prop allows handling `<button>`, `<input type="submit">`, and `<input type="image">` elements in a `<form>`.
- Custom Invocation with `startTransition`: Invoke Server Actions without using `action` or `formAction` by using `startTransition`. This method **disables Progressive Enhancement**.

action

You can use React's `action` prop to invoke a Server Action on a `form` element. Server Actions passed with the `action` prop act as asynchronous side effects in response to user interaction.

JS app/add-to-cart.js

```
1 export default function AddToCart({ productId }) {
2   async function addItem(data) {
3     'use server';
4
5     const cartId = cookies().get('cartId')?.value;
6     await saveToDb({ cartId, data });
7   }
8
9   return (
10    <form action={addItem}>
11      <button type="submit">Add to Cart</button>
12    </form>
13  );
14}
```

Note: An `action` is similar to the HTML primitive [action ↗](#)

formAction

You can use `formAction` prop to handle **Form Actions** on elements such as `button`, `input type="submit"`, and `input type="image"`. The `formAction` prop takes precedence over the form's `action`.

JS app/form

```
1 export default function Form() {
2   async function handleSubmit() {
3     'use server';
4     // ...
5   }
6
7   async function submitImage() {
8     'use server';
9     // ...
10}
```

```
11
12   return (
13     <form action={handleSubmit}>
14       <input type="text" name="name" />
15       <input type="image" formAction={submitImage} />
16       <button type="submit">Submit</button>
17     </form>
18   );
19 }
```

Note: A `formAction` is the HTML primitive [formaction](#). React now allows you to pass functions to this attribute.

Custom invocation using `startTransition`

You can also invoke Server Actions without using `action` or `formAction`. You can achieve this by using `startTransition` provided by the `useTransition` hook, which can be useful if you want to use Server Actions outside of forms, buttons, or inputs.

Note: Using `startTransition` disables the out-of-the-box Progressive Enhancement.

JS app/components/example-client-component.js

```
1 'use client';
2
3 import { useTransition } from 'react';
4 import { addItem } from '../actions';
5
6 function ExampleClientComponent({ id }) {
7   let [isPending, startTransition] = useTransition();
8
9   return (
10     <button onClick={() => startTransition(() => addItem(id))}>
11       Add To Cart
12     </button>
13   );
14 }
```

JS app/actions.js

```
1 'use server';
2
3 export async function addItem(id) {
4   await addItemToDb(id);
5   // Marks all product pages for revalidating
6   revalidatePath('/product/[id]');
7 }
```

Custom invocation without `startTransition`

If you aren't doing [Server Mutations](#), you can directly pass the function as a prop like any other function.

TS app/posts/[id]/page.tsx

```
1 import kv from '@vercel/kv';
2 import LikeButton from './like-button';
3
4 export default function Page({ params }: { params: { id: string } }) {
5   async function increment() {
6     'use server';
7     await kv.incr(`post:id:${params.id}`);
8   }
9
10  return <LikeButton increment={increment} />;
11 }
```

TS app/post/[id]/like-button.tsx

```
1 'use client';
2
3 export default function LikeButton({
4   increment,
5 }: {
6   increment: () => Promise<void>;
7 }) {
8   return (
9     <button
10       onClick={async () => {
11         await increment();
12       }}
13     >
14     Like
15     </button>
16   );
17 }
```

Enhancements

Experimental `useOptimistic`

The experimental `useOptimistic` hook provides a way to implement optimistic updates in your application. Optimistic updates are a technique that enhances user experience by making the app appear more responsive.

When a Server Action is invoked, the UI is updated immediately to reflect the expected outcome, instead of waiting for the Server Action's response.



```

1 'use client';
2
3 import { experimental_useOptimistic as useOptimistic } from 'react';
4 import { send } from './actions.js';
5
6 export function Thread({ messages }) {
7   const [optimisticMessages, addOptimisticMessage] = useOptimistic(
8     messages,
9     (state, newMessage) => [...state, { message: newMessage, sending: true }],
10   );
11   const formRef = useRef();
12
13   return (
14     <div>
15       {optimisticMessages.map((m) => (
16         <div>
17           {m.message}
18           {m.sending ? 'Sending...' : ''}
19         </div>
20       ))}
21     <form
22       action={async (formData) => {
23         const message = formData.get('message');
24         formRef.current.reset();
25         addOptimisticMessage(message);
26         await send(message);
27       }}
28       ref={formRef}
29     >
30       <input type="text" name="message" />
31     </form>
32   </div>
33 );
34 }

```

Experimental useFormStatus

The **experimental** `useFormStatus` hook can be used within Form Actions, and provides the `pending` property.



```

1 import { experimental_useFormStatus as useFormStatus } from 'react-dom';
2
3 function Submit() {
4   const { pending } = useFormStatus();
5
6   return (
7     <input

```

```
8     type="submit"
9     className={pending ? 'button-pending' : 'button-normal'}
10    disabled={pending}
11    >
12      Submit
13    </input>
14  );
15 }
```

Progressive Enhancement

Progressive Enhancement allows a `<form>` to function properly without JavaScript, or with JavaScript disabled. This allows users to interact with the form and submit data even if the JavaScript for the form hasn't been loaded yet or if it fails to load.

Both Server Form Actions and Client Form Actions support Progressive Enhancement, using one of two strategies:

- If a **Server Action** is passed directly to a `<form>`, the form is interactive **even if JavaScript is disabled**.
- If a **Client Action** is passed to a `<form>`, the form is still interactive, but the action will be placed in a queue until the form has hydrated. The `<form>` is prioritized with Selective Hydration, so it happens quickly.

js app/components/example-client-component.js

```
1 'use client';
2
3 import { useState } from 'react';
4 import { handleSubmit } from './actions.js';
5
6 export default function ExampleClientComponent({ myAction }) {
7   const [input, setInput] = useState();
8
9   return (
10     <form action={handleSubmit} onChange={(e) => setInput(e.target.value)}>
11       {/* ... */}
12     </form>
13   );
14 }
```

In both cases, the form is interactive before hydration occurs. Although Server Actions have the additional benefit of not relying on client JavaScript, you can still compose additional behavior with Client Actions where desired without sacrificing interactivity.

Examples

Usage with Client Components

Import

Server Actions **cannot** be *defined* within Client Components, but they can be *imported*. To use Server Actions in Client Components, you can import the action from a file containing a top-level `"use server"` directive.

JS app/actions.js

```
1 'use server';
2
3 export async function addItem() {
4   // ...
5 }
```

JS app/components/example-client-component.js

```
1 'use client';
2
3 import { useTransition } from 'react';
4 import { addItem } from '../actions';
5
6 function ExampleClientComponent({ id }) {
7   let [isPending, startTransition] = useTransition();
8
9   return (
10     <button onClick={() => startTransition(() => addItem(id))}>
11       Add To Cart
12     </button>
13   );
14 }
```

Props

Although [importing](#) Server Actions is recommended, in some cases you might want to pass down a Server Action to a Client Component as a prop.

For example, you might want to use a dynamically generated value within the action. In that case, passing a Server Action down as a prop might be a viable solution.

JS app/components/example-server-component.js

```
1 import { ExampleClientComponent } from './components/example-client-component.js';
2
3 function ExampleServerComponent({ id }) {
4   async function updateItem(data) {
5     'use server';
6     modifyItem({ id, data });
7   }
8
9   return <ExampleClientComponent updateItem={updateItem} />;
10 }
```

Js app/components/example-client-component.js

```
1 'use client';
2
3 function ExampleClientComponent({ updateItem }) {
4   async function action(formData: FormData) {
5     await updateItem(formData);
6   }
7
8   return (
9     <form action={action}>
10       <input type="text" name="name" />
11       <button type="submit">Update Item</button>
12     </form>
13   );
14 }
```



On-demand Revalidation

Server Actions can be used to revalidate data on-demand by path ([revalidatePath](#)) or by cache tag ([revalidateTag](#)).

```
1 import { revalidateTag } from 'next/cache';
2
3 async function revalidate() {
4   'use server';
5   revalidateTag('blog-posts');
6 }
```

Validation

The data passed to a Server Action can be validated or sanitized before invoking the action. For example, you can create a wrapper function that receives the action as its argument, and returns a function that invokes the action if it's valid.



```

1 'use server';
2
3 import { withValidate } from 'lib/form-validation';
4
5 export const action = withValidate((data) => {
6   // ...
7 });

```



```

1 export function withValidate(action) {
2   return (formData: FormData) => {
3     'use server';
4
5     const isValidData = verifyData(formData);
6
7     if (!isValidData) {
8       throw new Error('Invalid input.');
9     }
10
11    const data = process(formData);
12    return action(data);
13  };
14 }

```

Using headers

You can read incoming request headers such as `cookies` and `headers` within a Server Action.

```

1 import { cookies } from 'next/headers';
2
3 async function addItem(data) {
4   'use server';
5
6   const cartId = cookies().get('cartId')?.value;
7
8   await saveToDb({ cartId, data });
9 }

```

Additionally, you can modify cookies within a Server Action.

```

1 import { cookies } from 'next/headers';
2
3 async function create(data) {
4   'use server';

```

```
5
6  const cart = await createCart():
7  cookies().set('cartId', cart.id)
8  // or
9  cookies().set({
10    name: 'cartId',
11    value: cart.id,
12    httpOnly: true,
13    path: '/'
14  })
15 }
```

Glossary

Actions

Performs asynchronous side effects in response to user interaction, with built-in solutions for error handling and [optimistic updates](#). Similar to the HTML primitive [action](#) ↗.

Form Actions

[Actions](#) integrated into the web standard `<form>` API, and enable out-of-the-box progressive enhancement and [loading states](#). Similar to the HTML primitive [formaction](#) ↗.

Server Functions

Functions that run on the server, but can be called on the client.

Server Actions

[Server Functions](#) called as an action.

Server Mutations

[Server Actions](#) that mutates your data and calls `redirect`, `revalidatePath`, or `revalidateTag`.

Next Steps

For more information on what to do next, we recommend the following sections

cookies

API Reference for the cookies function.

> Menu

App 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 }
```

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.

> Menu

App 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.

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.

Supported Features

The core of Next.js has been designed to support static exports.

Server Components

When you run `next build` to generate a static export, Server Components consumed inside the `app` directory will run during the build, similar to traditional static-site generation.

The resulting component will be rendered into static HTML for the initial page load and a static payload for client navigation between routes. No changes are required for your Server Components when using the static export, unless they consume [dynamic server functions](#).

TS app/page.tsx

```
1 export default async function Page() {
2   // This fetch will run on the server during `next build`
3   const res = await fetch('https://api.example.com/...');
4   const data = await res.json();
5
6   return <main>...</main>;
7 }
```

Client Components

If you want to perform data fetching on the client, you can use a Client Component with [SWR](#) to deduplicate requests.

TS app/other/page.tsx

```
1 'use client';
2
3 import useSWR from 'swr';
4
5 const fetcher = (url: string) => fetch(url).then((r) => r.json());
6
7 export default function Page() {
8   const { data, error } = useSWR(
9     `https://jsonplaceholder.typicode.com/posts/1`,
10     fetcher,
11   );
12   if (error) return 'Failed to load';
13   if (!data) return 'Loading...';
14
15   return data.title;
```

Since route transitions happen client-side, this behaves like a traditional SPA. For example, the following index route allows you to navigate to different posts on the client:

TS app/page.tsx

```

1 import Link from 'next/link';
2
3 export default function Page() {
4   return (
5     <>
6       <h1>Index Page</h1>
7       <hr />
8       <ul>
9         <li>
10          <Link href="/post/1">Post 1</Link>
11        </li>
12        <li>
13          <Link href="/post/2">Post 2</Link>
14        </li>
15      </ul>
16    </>
17  );
18}

```

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 }

```

Route Handlers

Route Handlers will render a static response when running `next build`. Only the `GET` HTTP verb is supported. This can be used to generate static HTML, JSON, TXT, or other files from dynamic or static data. For example:

```

1 import { NextResponse } from 'next/server';
2
3 export async function GET() {
4   return NextResponse.json({ name: 'Lee' });
5 }

```

The above file `app/data.json/route.ts` will render to a static file during `next build`, producing `data.json` containing `{ name: 'Lee' }`.

If you need to read dynamic values from the incoming request, you cannot use a static export.

Browser APIs

Client Components are pre-rendered to HTML during `next build`. Because Web APIs ↗ like `window`, `localStorage`, and `navigator` are not available on the server, you need to safely access these APIs only when running in the browser. For example:

```
1  'use client';
2
3  import { useEffect } from 'react';
4
5  export default function ClientComponent() {
6    useEffect(() => {
7      // You now have access to `window`
8      console.log(window.innerHeight);
9    }, [])
10
11  return ...;
12 }
```

Unsupported Features

After enabling the static export `output` mode, all routes inside `app` are opted-into the following [Route Segment Config](#):

```
export const dynamic = 'error';
```

With this configuration, your application **will produce an error** if you try to use server functions like `headers` or `cookies`, since there is no runtime server. This ensures local development matches the same behavior as a static export. If you need to use server functions, you cannot use a static export.

The following additional dynamic features are not supported with a static export:

- `rewrites` in `next.config.js`
- `redirects` in `next.config.js`
- `headers` in `next.config.js`
- Middleware
- [Incremental Static Regeneration](#)

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

App 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 Metadata API in Next.js allows you to modify the `<head>` element of a page. You can configure metadata in two ways:

- **Config-based Metadata:** Export a static metadata object or a dynamic generateMetadata function in a `layout.js` or `page.js` file.
- **File-based Metadata:** Add static or dynamically generated special files to route segments.

Additionally, you can create dynamic Open Graph Images using JSX and CSS with [imageResponse](#) constructor.

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 [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.

Metadata

Use the Metadata API to define metadata in any layout or page.

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

> Menu

App 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.

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 ↗ \(INP\) \(*experimental*\)](#)

> Menu

App 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.

TS app/layout.tsx

▼



```
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({
5   subsets: ['latin'],
6   display: 'swap',
7 });
8
```

```
9 export default function RootLayout({
10   children,
11 }: {
12   children: React.ReactNode;
13 }) {
14   return (
15     <html lang="en" className={inter.className}>
16       <body>{children}</body>
17     </html>
18   );
19 }
```

If you can't use a variable font, you will **need to specify a weight**:

TS app/layout.tsx

```
1 import { Roboto } from 'next/font/google';
2
3 const roboto = Roboto({
4   weight: '400',
5   subsets: ['latin'],
6   display: 'swap',
7 });
8
9 export default function RootLayout({
10   children,
11 }: {
12   children: React.ReactNode;
13 }) {
14   return (
15     <html lang="en" className={roboto.className}>
16       <body>{children}</body>
17     </html>
18   );
19 }
```

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

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:

TS app/layout.tsx

```
const inter = Inter({ subsets: ['latin'] });
```

View the [Font API Reference](#) for more information.

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:

TS 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});
```

TS app/layout.tsx

```
1 import { inter } from './fonts';
2
3 export default function Layout({ children }: { children: React.ReactNode }) {
4   return (
5     <html lang="en" className={inter.className}>
6       <body>
```

```
7     <div>{children}</div>
8   </body>
9 </html>
10 );
11 }
```

TS app/page.tsx

```
1 import { roboto_mono } from './fonts';
2
3 export default function Page() {
4   return (
5     <>
6       <h1 className={roboto_mono.className}>My page</h1>
7     </>
8   );
9 }
```

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:

TS app/layout.tsx

```
1 import { Inter, Roboto_Mono } from 'next/font/google';
2 import styles from './global.css';
3
4 const inter = Inter({
5   subsets: ['latin'],
6   variable: '--font-inter',
7   display: 'swap',
8 });
9
10 const roboto_mono = Roboto_Mono({
11   subsets: ['latin'],
12   variable: '--font-roboto-mono',
13   display: 'swap',
14 });
15
16 export default function RootLayout({
17   children,
18 }: {
19   children: React.ReactNode;
20 }) {
21   return (
22     <html lang="en" className={`${inter.variable} ${roboto_mono.variable}`}>
23       <body>
24         <h1>My App</h1>
25         <div>{children}</div>
```

```
26      </body>
27    </html>
28  );
29 }
```

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.

app/layout.tsx



```
1 import localFont from 'next/font/local';
2
3 // Font files can be colocated inside of `app`
4 const myFont = localFont({
5   src: './my-font.woff2',
6   display: 'swap',
7 });
8
9 export default function RootLayout({
10   children,
11 }: {
12   children: React.ReactNode;
13 }) {
14   return (
15     <html lang="en" className={myFont.className}>
16       <body>{children}</body>
17     </html>
18   );
}
```

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 });

```

View the [Font API Reference](#) for more information.

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.

app/layout.tsx

```

1 import { Inter, Roboto_Mono } from 'next/font/google';
2
3 const inter = Inter({
4   subsets: ['latin'],

```

```
5   display: 'swap',
6   variable: '--font-inter',
7 });
8
9 const roboto_mono = Roboto_Mono({
10   subsets: ['latin'],
11   display: 'swap',
12   variable: '--font-roboto-mono',
13 });
14
15 export default function RootLayout({
16   children,
17 }: {
18   children: React.ReactNode;
19 }) {
20   return (
21     <html lang="en" className={`${inter.variable} ${roboto_mono.variable}`}>
22       <body>{children}</body>
23     </html>
24   );
25 }
```

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: [],
16 };
```

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 routes 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 a [layout](#), it is preloaded on all the routes wrapped by the layout.
 - If it's the [root layout](#), it is preloaded on all routes.
-

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

API Reference

Learn more about the next/font API.

App Router > ... > Components

Font

Optimizing loading web fonts with the built-in `next/font` loaders.

> Menu

App 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 app/page.js



```
1 import Image from 'next/image';
2 import profilePic from './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 import profilePic from '../public/me.png';
3
4 export default function Page() {
5   return <Image src={profilePic} alt="Picture of the author" priority />;
6 }
```

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.](#)

API Reference

Learn more about the `next/image` API.

App Router > ... > Components

[`<Image>`](#)

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

> Menu

App 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

App 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

Importing Client Components

`js app/page.js`

```
1 'use client';
2
3 import { useState } from 'react';
4 import dynamic from 'next/dynamic';
5
6 // Client Components:
7 const ComponentA = dynamic(() => import('../components/A'));
8 const ComponentB = dynamic(() => import('../components/B'));
9 const ComponentC = dynamic(() => import('../components/C'), { ssr: false });
10
11 export default function ClientComponentExample() {
12   const [showMore, setShowMore] = useState(false);
13
14   return (
15     <div>
16       {/* Load immediately, but in a separate client bundle */}
17       <ComponentA />
18
19       {/* Load on demand, only when/if the condition is met */}
20       {showMore && <ComponentB />}
21       <button onClick={() => setShowMore(!showMore)}>Toggle</button>
22
23       {/* Load only on the client side */}
24       <ComponentC />
25     </div>
26   );
27 }
```

Skiping SSR

When using `React.lazy()` and Suspense, Client Components will be pre-rendered (SSR) by default.

If you want to disable pre-rendering for a Client Component, you can use the `ssr` option set to `false`:

```
const ComponentC = dynamic(() => import('../components/C'), { ssr: false });
```

Importing Server Components

If you dynamically import a Server Component, only the Client Components that are children of the Server Component will be lazy-loaded - not the Server Component itself.

JS app/page.js



```
1 import dynamic from 'next/dynamic';
2
3 // Server Component:
4 const ServerComponent = dynamic(() => import('../components/ServerComponent'));
5
```

```
6 export default function ServerComponentExample() {
7   return (
8     <div>
9       <ServerComponent />
10    </div>
11  );
12}
```

Loading External Libraries

External libraries can be loaded on demand using the `import()` [function](#). This example uses the external library `fuse.js` for fuzzy search. The module is only loaded on the client after the user types in the search input.

js app/page.js



```
1 'use client';
2
3 import { useState } from 'react';
4
5 const names = ['Tim', 'Joe', 'Bel', 'Lee'];
6
7 export default function Page() {
8   const [results, setResults] = useState();
9
10  return (
11    <div>
12      <input
13        type="text"
14        placeholder="Search"
15        onChange={async (e) => {
16          const { value } = e.currentTarget;
17          // Dynamically load fuse.js
18          const Fuse = (await import('fuse.js')).default;
19          const fuse = new Fuse(names);
20
21          setResults(fuse.search(value));
22        }}
23      />
24      <pre>Results: {JSON.stringify(results, null, 2)}</pre>
25    </div>
26  );
27}
```

Adding a custom loading component

js app/page.js



```
1 import dynamic from 'next/dynamic';
2
3 const WithCustomLoading = dynamic(
4   () => import('../components/WithCustomLoading'),
5   {
6     loading: () => <p>Loading...</p>,
7   },
8 );
9
10 export default function Page() {
11   return (
12     <div>
13       /* The loading component will be rendered while <WithCustomLoading/> is loading */
14       <WithCustomLoading />
15     </div>
16   );
17 }
```

Importing Named Exports

To dynamically import a named export, you can return it from the Promise returned by `import()` ↗ function:

JS components/hello.js

```
1 'use client';
2
3 export function Hello() {
4   return <p>Hello!</p>;
5 }
```

JS app/page.js

```
1 import dynamic from 'next/dynamic';
2
3 const ClientComponent = dynamic(() =>
4   import('../components/ClientComponent').then((mod) => mod.Hello),
5 );
```

> Menu

App Router > ... > Optimizing > Metadata

Metadata

Next.js has a Metadata API that can be used to define your application metadata (e.g. `meta` and `link` tags inside your HTML `head` element) for improved SEO and web shareability.

There are two ways you can add metadata to your application:

- **Config-based Metadata:** Export a static `metadata` object or a dynamic `generateMetadata` function in a `layout.js` or `page.js` file.
- **File-based Metadata:** Add static or dynamically generated special files to route segments.

With both these options, Next.js will automatically generate the relevant `<head>` elements for your pages.

You can also create dynamic OG images using the `ImageResponse` constructor.

Static Metadata

To define static metadata, export a `Metadata` object from a `layout.js` or static `page.js` file.

TS layout.tsx / page.tsx

▼



```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: '...',
5   description: '...',
6 };
7
8 export default function Page() {}
```

For all the available options, see the [API Reference](#).

Dynamic Metadata

You can use `generateMetadata` function to `fetch` metadata that requires dynamic values.

```
ts app/products/[id]/page.tsx

1 import { Metadata, ResolvingMetadata } from 'next';
2
3 type Props = {
4   params: { id: string };
5   searchParams: { [key: string]: string | string[] | undefined };
6 };
7
8 export async function generateMetadata(
9   { params, searchParams }: Props,
10   parent?: ResolvingMetadata,
11 ): Promise<Metadata> {
12   // read route params
13   const id = params.id;
14
15   // fetch data
16   const product = await fetch(`https://.../${id}`).then((res) => res.json());
17
18   // optionally access and extend (rather than replace) parent metadata
19   const previousImages = (await parent).openGraph?.images || [];
20
21   return {
22     title: product.title,
23     openGraph: {
24       images: ['/some-specific-page-image.jpg', ...previousImages],
25     },
26   };
27 }
28
29 export default function Page({ params, searchParams }: Props) {}
```

For all the available params, see the [API Reference](#).

Good to know:

- Both static and dynamic metadata through `generateMetadata` are **only supported in Server Components**.
- When rendering a route, Next.js will automatically deduplicate `fetch` requests for the same data across `generateMetadata`, `generateStaticParams`, Layouts, Pages, and Server Components. React `cache` can be used if `fetch` is unavailable.
- Next.js will wait for data fetching inside `generateMetadata` to complete before streaming UI to the client. This guarantees the first part of a `streamed response` includes `<head>` tags.

File-based metadata

These special files are available for metadata:

- [favicon.ico, apple-icon.jpg, and icon.jpg](#)
- [opengraph-image.jpg and twitter-image.jpg](#)
- [robots.txt](#)
- [sitemap.xml](#)

You can use these for static metadata, or you can programmatically generate these files with code.

For implementation and examples, see the [Metadata Files API Reference](#) and [Dynamic Image Generation](#).

Behavior

File-based metadata has the higher priority and will override any config-based metadata.

Default Fields

There are two default `meta` tags that are always added even if a route doesn't define metadata:

- The [meta charset tag ↗](#) sets the character encoding for the website.
- The [meta viewport tag ↗](#) sets the viewport width and scale for the website to adjust for different devices.

```
1 <meta charset="utf-8" />
2 <meta name="viewport" content="width=device-width, initial-scale=1" />
```

Note: You can overwrite the default `viewport` meta tag.

Ordering

Metadata is evaluated in order, starting from the root segment down to the segment closest to the final `page.js` segment. For example:

1. `app/layout.tsx` (Root Layout)

2. `app/blog/layout.tsx` (Nested Blog Layout)

3. `app/blog/[slug]/page.tsx` (Blog Page)

Merging

Following the [evaluation order](#), Metadata objects exported from multiple segments in the same route are **shallowly** merged together to form the final metadata output of a route. Duplicate keys are **replaced** based on their ordering.

This means metadata with nested fields such as `openGraph` and `robots` that are defined in an earlier segment are **overwritten** by the last segment to define them.

Overwriting fields

`js app/layout.js`

```
1 export const metadata = {
2   title: 'Acme',
3   openGraph: {
4     title: 'Acme',
5     description: 'Acme is a...',
6   },
7 };
```

`js app/blog/page.js`

```
1 export const metadata = {
2   title: 'Blog',
3   openGraph: {
4     title: 'Blog',
5   },
6 };
7
8 // Output:
9 // <title>Blog</title>
10 // <meta property="og:title" content="Blog" />
```

In the example above:

- `title` from `app/layout.js` is **replaced** by `title` in `app/blog/page.js`.
- All `openGraph` fields from `app/layout.js` are **replaced** in `app/blog/page.js` because `app/blog/page.js` sets `openGraph` metadata. Note the absence of `openGraph.description`.

If you'd like to share some nested fields between segments while overwriting others, you can pull them out into a separate variable:

js app/shared-metadata.js

```
export const openGraphImage = { images: ['http://...'] };
```

js app/page.js

```
1 import { openGraphImage } from './shared-metadata';
2
3 export const metadata = {
4   openGraph: {
5     ...openGraphImage,
6     title: 'Home',
7   },
8 };
```

js app/about/page.js

```
1 import { openGraphImage } from '../shared-metadata';
2
3 export const metadata = {
4   openGraph: {
5     ...openGraphImage,
6     title: 'About',
7   },
8 };
```

In the example above, the OG image is shared between `app/layout.js` and `app/about/page.js` while the titles are different.

Inheriting fields

js app/layout.js

```
1 export const metadata = {
2   title: 'Acme',
3   openGraph: {
4     title: 'Acme',
5     description: 'Acme is a...',
6   },
7 };
```

js app/about/page.js

```
1 export const metadata = {
2   title: 'About',
```

```
3  };
4
5 // Output:
6 // <title>About</title>
7 // <meta property="og:title" content="Acme" />
8 // <meta property="og:description" content="Acme is a..." />
```

Notes

- title from app/layout.js is replaced by title in app/about/page.js.
- All openGraph fields from app/layout.js are inherited in app/about/page.js because app/about/page.js doesn't set openGraph metadata.

Dynamic Image Generation

The ImageResponse constructor allows you to generate dynamic images using JSX and CSS. This is useful for creating social media images such as Open Graph images, Twitter cards, and more.

ImageResponse uses the Edge Runtime, and Next.js automatically adds the correct headers to cached images at the edge, helping improve performance and reducing recomputation.

To use it, you can import ImageResponse from next/server :

app/about/route.jsx



```
1 import { ImageResponse } from 'next/server';
2
3 export const runtime = 'edge';
4
5 export async function GET() {
6   return new ImageResponse(
7     (
8       <div
9         style={{
10           fontSize: 128,
11           background: 'white',
12           width: '100%',
13           height: '100%',
14           display: 'flex',
15           textAlign: 'center',
16           alignItems: 'center',
17           justifyContent: 'center',
18         }}
19       >
20         Hello world!
21       </div>
22     )
23   )
24 }
```

```
21     </div>
22   ),
23   {
24     width: 1200,
25     height: 600,
26   },
27 );
28 }
```

`ImageResponse` integrates well with other Next.js APIs, including [Route Handlers](#) and file-based Metadata. For example, you can use `ImageResponse` in a `opengraph-image.tsx` file to generate Open Graph images at build time or dynamically at request time.

`ImageResponse` supports common CSS properties including flexbox and absolute positioning, custom fonts, text wrapping, centering, and nested images. [See the full list of supported CSS properties](#).

Good to know:

- Examples are available in the [Vercel OG Playground](#).
- `ImageResponse` uses [@vercel/og](#), [Satori](#), and Resvg to convert HTML and CSS into PNG.
- Only the Edge Runtime is supported. The default Node.js runtime will not work.
- Only flexbox and a subset of CSS properties are supported. Advanced layouts (e.g. `display: grid`) will not work.
- Maximum bundle size of `500KB`. The bundle size includes your JSX, CSS, fonts, images, and any other assets. If you exceed the limit, consider reducing the size of any assets or fetching at runtime.
- Only `ttf`, `otf`, and `woff` font formats are supported. To maximize the font parsing speed, `ttf` or `otf` are preferred over `woff`.

JSON-LD

[JSON-LD](#) is a format for structured data that can be used by search engines to understand your content. For example, you can use it to describe a person, an event, an organization, a movie, a book, a recipe, and many other types of entities.

Our current recommendation for JSON-LD is to render structured data as a `<script>` tag in your `layout.js` or `page.js` components. For example:

TS app/products/[id]/page.tsx

```
1 export default async function Page({ params }) {
2   const product = await getProduct(params.id);
3
4   const jsonLd = {
```

```
5   '@context': 'https://schema.org',
6   '@type': 'Product',
7   name: product.name,
8   image: product.image,
9   description: product.description,
10  };
11
12  return (
13    <section>
14      {/* Add JSON-LD to your page */}
15      <script
16          type="application/ld+json"
17          dangerouslySetInnerHTML={{ __html: JSON.stringify(jsonLd) }}
18      />
19      {/* ... */}
20    </section>
21  );
22 }
```

You can validate and test your structured data with the [Rich Results Test ↗](#) for Google or the generic [Schema Markup Validator ↗](#).

You can type your JSON-LD with TypeScript using community packages like [schema-dts ↗](#):

```
1 import { Product, WithContext } from 'schema-dts';
2
3 const jsonLd: WithContext<Product> = {
4   '@context': 'https://schema.org',
5   '@type': 'Product',
6   name: 'Next.js Sticker',
7   image: 'https://nextjs.org/imgs/sticker.png',
8   description: 'Dynamic at the speed of static.',
9 }
```

Next Steps

View all the Metadata API options.

App Router > ... > Functions

[generateMetadata](#)

Learn how to add Metadata to your Next.js application for improved search engine optimization (SEO) and web shareability.

App Router > ... > File Conventions

Metadata Files

API documentation for the metadata file conventions.

> Menu

App 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

App Router > ... > Optimizing > Scripts

Script Optimization

Layout Scripts

To load a third-party script for multiple routes, import `next/script` and include the script directly in your layout component:

TS app/dashboard/layout.tsx

```
1 import Script from 'next/script';
2
3 export default function DashboardLayout({
4   children,
5 }: {
6   children: React.ReactNode;
7 }) {
8   return (
9     <>
10       <section>{children}</section>
11       <Script src="https://example.com/script.js" />
12     </>
13   );
14 }
```

The third-party script is fetched when the the folder route (e.g. `dashboard/page.js`) or any nested route (e.g. `dashboard/settings/page.js`) is accessed by the user. Next.js will ensure the script will **only load once**, even if a user navigates between multiple routes in the same layout.

Application Scripts

To load a third-party script for all routes, import `next/script` and include the script directly in your root layout:

TS app/layout.tsx

```
1 import Script from 'next/script';
2
```

```
3 export default function RootLayout({  
4   children,  
5 }: {  
6   children: React.ReactNode;  
7 }) {  
8   return (  
9     <html lang="en">  
10       <body>{children}</body>  
11       <Script src="https://example.com/script.js" />  
12     </html>  
13   );  
14 }
```

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`:

```
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 }
```

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:

TS app/page.tsx



```
1 'use client';
2
3 import Script from 'next/script';
4
5 export default function Page() {
6   return (
7     <>
8       <Script
9         src="https://example.com/script.js"
10        onLoad={() => {
11          console.log('Script has loaded');
12        }}
13      />
14    </>
15  );
16}
```

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.

TS app/page.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}
```

API Reference

Learn more about the `next/script` API.

<Script>

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

> Menu

App 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

App Router > Building Your Application > Rendering

Rendering

Rendering converts the code you write into user interfaces.

React 18 and Next.js 13 introduced new ways to render your application. This page will help you understand the differences between rendering environments, strategies, runtimes, and how to opt into them.

Rendering Environments

There are two environments where your application code can be rendered: the client and the server.

- The **client** refers to the browser on a user's device that sends a request to a server for your application code. It then turns the response from the server into an interface the user can interact with.
- The **server** refers to the computer in a data center that stores your application code, receives requests from a client, does some computation, and sends back an appropriate response.

Note: Server can refer to computers in regions where your application is deployed to, the [Edge Network](#) ↗ where your application code is distributed, or [Content Delivery Networks \(CDNs\)](#) ↗ where the result of the rendering work can be cached.

Component-level Client and Server Rendering

Before React 18, the primary way to render your application **using React** was entirely on the client.

Next.js provided an easier way to break down your application into **pages** and prerender on the server by generating HTML and sending it to the client to be [hydrated](#) ↗ by React. However, this led to additional JavaScript needed on the client to make the initial HTML interactive.

Now, with [Server and Client Components](#), React can render on the client **and** the server meaning you can choose the rendering environment at the component level.

By default, the [`app`](#) router uses **Server Components**, allowing you to easily render components on the server and reducing the amount of JavaScript sent to the client.

Static and Dynamic Rendering on the Server

In addition to client-side and server-side rendering with React components, Next.js gives you the option to optimize rendering on the server with **Static** and **Dynamic** Rendering.

Static Rendering

With **Static Rendering**, both Server and Client Components can be prerendered on the server at **build time**. The result of the work is [cached](#) and reused on subsequent requests. The cached result can also be [revalidated](#).

Note: This is equivalent to [Static Site Generation \(SSG\)](#) and [Incremental Static Regeneration \(ISR\)](#) in the Pages Router.

Server and Client Components are rendered differently during Static Rendering:

- Client Components have their HTML and JSON prerendered and cached on the server. The cached result is then sent to the client for hydration.

- Server Components are rendered on the server by React, and their payload is used to generate HTML. The same rendered payload is also used to hydrate the components on the client, resulting in no JavaScript needed on the client.

Dynamic Rendering

With Dynamic Rendering, both Server *and* Client Components are rendered on the server at **request time**. The result of the work is not cached.

Note: This is equivalent to [Server-Side Rendering \(`getServerSideProps\(\)` \)](#) in the [Pages Router](#).

To learn more about static and dynamic behavior, see the [Static and Dynamic Rendering](#) page. To learn more about caching, see the [Caching](#) sections.

Edge and Node.js Runtimes

On the server, there are two runtimes where your pages can be rendered:

- The **Node.js Runtime** (default) has access to all Node.js APIs and compatible packages from the ecosystem.
- The **Edge Runtime** is based on [Web APIs](#).

Both runtimes support [streaming](#) from the server, depending on your deployment infrastructure.

To learn how to switch between runtimes, see the [Edge and Node.js Runtimes](#) page.

Next Steps

Now that you understand the fundamentals of rendering, you can learn more about implementing the different rendering strategies and runtimes:

[Static and Dynamic](#)

Learn about static and dynamic rendering in Next.js.

Edge and Node.js Runtimes

Learn about the switchable runtimes (Edge and Node.js) in Next.js.

> Menu

App 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".

Examples

Segment Runtime Option

You can specify a runtime for individual route segments in your Next.js application. To do so, declare a variable called `runtime` and export it. The variable must be a string, and must have a value of either '`nodejs`' or '`edge`' runtime.

The following example demonstrates a page route segment that exports a `runtime` with a value of '`edge`' :

```
TS app/page.tsx
```

```
export const runtime = 'edge'; // 'nodejs' (default) | 'edge'
```

If the segment `runtime` is *not* set, the default `nodejs` runtime will be used. You do not need to use the `runtime` option if you do not plan to change from the Node.js runtime.

Static and Dynamic Rendering

In Next.js, a route can be statically or dynamically rendered.

- In a **static** route, components are rendered on the server at build time. The result of the work is cached and reused on subsequent requests.
- In a **dynamic** route, components are rendered on the server at request time.

Static Rendering (Default)

By default, Next.js statically renders routes to improve performance. This means all the rendering work is done ahead of time and can be served from a Content Delivery Network (CDN) geographically closer to the user.

Static Data Fetching (Default)

By default, Next.js will cache the result of `fetch()` requests that do not specifically opt out of caching behavior. This means that fetch requests that do not set a `cache` option will use the `force-cache` option.

If any fetch requests in the route use the `revalidate` option, the route will be re-rendered statically during revalidation.

To learn more about caching data fetching requests, see the [Caching and Revalidating](#) page.

Dynamic Rendering

During static rendering, if a dynamic function or a dynamic `fetch()` request (no caching) is discovered, Next.js will switch to dynamically rendering the whole route at request time. Any cached data requests can still be re-used during dynamic rendering.

This table summarizes how [dynamic functions](#) and [caching](#) affect the rendering behavior of a route:

Data Fetching	Dynamic Functions	Rendering
Static (Cached)	No	Static
Static (Cached)	Yes	Dynamic
Not Cached	No	Dynamic
Not Cached	Yes	Dynamic

Note how dynamic functions always opt the route into dynamic rendering, regardless of whether the data fetching is cached or not. In other words, static rendering is dependent not only on the data fetching behavior, but also on the dynamic functions used in the route.

Note: In the future, Next.js will introduce hybrid server-side rendering where layouts and pages in a route can be independently statically or dynamically rendered, instead of the whole route.

Dynamic Functions

Dynamic functions rely on information that can only be known at request time such as a user's cookies, current requests headers, or the URL's search params. In Next.js, these dynamic functions are:

- Using `cookies()` or `headers()` in a Server Component will opt the whole route into dynamic rendering at request time.
- Using `useSearchParams()` in Client Components will skip static rendering and instead render all Client Components up to the nearest parent Suspense boundary on the client.
- We recommend wrapping the Client Component that uses `useSearchParams()` in a `<Suspense>` boundary. This will allow any Client Components above it to be statically rendered. [Example](#).
- Using the `searchParams` [Pages](#) prop will opt the page into dynamic rendering at request time.

Dynamic Data Fetching

Dynamic data fetches are `fetch()` requests that specifically opt out of caching behavior by setting the `cache` option to `'no-store'` or `revalidate` to `0`.

The caching options for all `fetch` requests in a layout or page can also be set using the [segment config](#) object.

To learn more about Dynamic Data Fetching, see the [Data Fetching](#) page.

> Menu

App Router > Building Your Application > Routing

Routing Fundamentals

The skeleton of every application is routing. This page will introduce you to the **fundamental concepts** of routing for the web and how to handle routing in Next.js.

Terminology

First, you will see these terms being used throughout the documentation. Here's a quick reference:

- **Tree:** A convention for visualizing a hierarchical structure. For example, a component tree with parent and children components, a folder structure, etc.
- **Subtree:** Part of a tree, starting at a new root (first) and ending at the leaves (last).
- **Root:** The first node in a tree or subtree, such as a root layout.

- **Leaf:** Nodes in a subtree that have no children, such as the last segment in a URL path.

- **URL Segment:** Part of the URL path delimited by slashes.
- **URL Path:** Part of the URL that comes after the domain (composed of segments).

The `app` Directory

In version 13, Next.js introduced a new **App Router** built on [React Server Components](#), which supports shared layouts, nested routing, loading states, error handling, and more.

The App Router works in a new directory named `app`. The `app` directory works alongside the `pages` directory to allow for incremental adoption. This allows you to opt some routes of your application into the new behavior while keeping other routes in the `pages` directory for previous behavior. If your application uses the `pages` directory, please also see the [Pages Router](#) documentation.

Good to know: The App Router takes priority over the Pages Router. Routes across directories should not resolve to the same URL path and will cause a build-time error to prevent a conflict.

By default, components inside `app` are [React Server Components](#). This is a performance optimization and allows you to easily adopt them, and you can also use [Client Components](#).

Recommendation: Check out the [Server and Client Components](#) page if you're new to Server Components.

Roles of Folders and Files

Next.js uses a file-system based router where:

- **Folders** are used to define routes. A route is a single path of nested folders, following the file-system hierarchy from the **root folder** down to a final **leaf folder** that includes a `page.js` file. See [Defining Routes](#).
- **Files** are used to create UI that is shown for a route segment. See [special files](#).

Route Segments

Each folder in a route represents a **route segment**. Each route segment is mapped to a corresponding **segment** in a **URL path**.

Nested Routes

To create a nested route, you can nest folders inside each other. For example, you can add a new `/dashboard/settings` route by nesting two new folders in the `app` directory.

The `/dashboard/settings` route is composed of three segments:

- `/` (Root segment)
 - `dashboard` (Segment)
 - `settings` (Leaf segment)
-

File Conventions

Next.js provides a set of special files to create UI with specific behavior in nested routes:

<code>layout</code>	Shared UI for a segment and its children
<code>page</code>	Unique UI of a route and make routes publicly accessible
<code>loading</code>	Loading UI for a segment and its children
<code>not-found</code>	Not found UI for a segment and its children
<code>error</code>	Error UI for a segment and its children
<code>global-error</code>	Global Error UI
<code>route</code>	Server-side API endpoint
<code>template</code>	Specialized re-rendered Layout UI
<code>default</code>	Fallback UI for Parallel Routes

Good to know: `.js`, `.jsx`, or `.tsx` file extensions can be used for special files.

Component Hierarchy

The React components defined in special files of a route segment are rendered in a specific hierarchy:

- `layout.js`
- `template.js`
- `error.js` (React error boundary)

- `loading.js` (React suspense boundary)
- `not-found.js` (React error boundary)
- `page.js` or nested `layout.js`

In a nested route, the components of a segment will be nested **inside** the components of its parent segment.

Colocation

In addition to special files, you have the option to colocate your own files (e.g. components, styles, tests etc) inside folders in the `app` directory.

This is because while folders define routes, only the contents returned by `page.js` or `route.js` are publically addressable.

Learn more about [Project Organization and Colocation](#).

Server-Centric Routing with Client-side Navigation

Unlike the `pages` directory which uses client-side routing, the App Router uses **server-centric routing** to align with [Server Components](#) and [data fetching on the server](#). With server-centric routing, the client does not have to download a route map and the same request for Server Components can be used to look up routes. This optimization is useful for all applications, but has a larger impact on applications with many routes.

Although routing is server-centric, the router uses **client-side navigation** with the [Link Component](#) - resembling the behavior of a Single-Page Application. This means when a user navigates to a new route, the

browser will not reload the page. Instead, the URL will be updated and Next.js will [only render the segments that change](#).

Additionally, as users navigate around the app, the router will store the result of the React Server Component payload in an **in-memory client-side cache**. The cache is split by route segments which allows invalidation at any level and ensures consistency across [React's concurrent renders](#) ↗. This means that for certain cases, the cache of a previously fetched segment can be re-used, further improving performance.

Learn more about [Linking and Navigating](#).

Partial Rendering

When navigating between sibling routes (e.g. `/dashboard/settings` and `/dashboard/analytics` below), Next.js will only fetch and render the layouts and pages in routes that change. It will **not** re-fetch or re-render anything above the segments in the subtree. This means that in routes that share a layout, the layout will be preserved when a user navigates between sibling pages.

Without partial rendering, each navigation would cause the full page to re-render on the server. Rendering only the segment that's updating reduces the amount of data transferred and execution time, leading to improved performance.

Advanced Routing Patterns

The App Router also provides a set conventions to help you implement more advanced routing patterns.

These include:

- [Parallel Routes](#): Allow you to simultaneously show two or more pages in the same view that can be navigated independently. You can use them for split views that have their own sub-navigation. E.g. Dashboards.
- [Intercepting Routes](#): Allow you to intercept a route and show it in the context of another route. You can use these when keeping the context for the current page is important. E.g. Seeing all tasks while editing one task or expanding a photo in a feed.

These patterns allow you to build richer and more complex UIs, democratizing features that were historically complex for small teams and individual developers to implement.

Next Steps

Now that you understand the fundamentals of routing in Next.js, follow the links below to create your first routes:

[Defining Routes](#)

Learn how to create your first route in Next.js.

[Pages and Layouts](#)

Create your first page and shared layout with the App Router.

[Linking and Navigating](#)

Learn how navigation works in Next.js, and how to use the Link Component and `useRouter` hook.

Route Groups

Route Groups can be used to partition your Next.js application into different sections.

Dynamic Routes

Dynamic Routes can be used to programmatically generate route segments from dynamic data.

Loading UI and Streaming

Built on top of Suspense, Loading UI allows you to create a fallback for specific route segments, and automatically stream content as it becomes ready.

Error Handling

Handle runtime errors by automatically wrapping route segments and their nested children in a React Error Boundary.

Parallel Routes

Simultaneously render one or more pages in the same view that can be navigated independently. A pattern for highly dynamic applications.

Intercepting Routes

Use intercepting routes to load a new route within the current layout while masking the browser URL, useful for advanced routing patterns such as modals.

Route Handlers

Create custom request handlers for a given route using the Web's Request and Response APIs.

Middleware

Learn how to use Middleware to run code before a request is completed.

Project Organization

Learn how to organize your Next.js project and colocate files.

Internationalization

Add support for multiple languages with internationalized routing and localized content.

> Menu

App Router > ... > Routing > Project Organization

Project Organization and File Colocation

Apart from [routing folder and file conventions](#), Next.js is **unopinionated** about how you organize and colocate your project files.

This page shares default behavior and features you can use to organize your project.

- [Safe colocation by default](#)
- [Project organization features](#)
- [Project organization strategies](#)

Safe colocation by default

In the `app` directory, [nested folder hierarchy](#) defines route structure.

Each folder represents a route segment that is mapped to a corresponding segment in a URL path.

However, even though route structure is defined through folders, a route is **not publically accessible** until a `page.js` or `route.js` file is added to a route segment.

And, even when a route is made publically accessible, only the **content returned** by `page.js` or `route.js` is sent to the client.

This means that **project files** can be **safely colocated** inside route segments in the `app` directory without accidentally being routable.

Good to know:

- This is different from the `pages` directory, where any file in `pages` is considered a route.

- While you **can** collocate your project files in `app` you don't **have** to. If you prefer, you can [keep them outside the `app` directory](#).

Project organization features

Next.js provides several features to help you organize your project.

Private Folders

Private folders can be created by prefixing a folder with an underscore: `_folderName`

This indicates the folder is a private implementation detail and should not be considered by the routing system, thereby **opting the folder and all its subfolders** out of routing.

Since files in the `app` directory can be [safely colocated by default](#), private folders are not required for colocation. However, they can be useful for:

- Separating UI logic from routing logic.
- Consistently organizing internal files across a project and the Next.js ecosystem.
- Sorting and grouping files in code editors.
- Avoiding potential naming conflicts with future Next.js file conventions.

Good to know

- While not a framework convention, you might also consider marking files outside private folders as "private" using the same underscore pattern.
- You can create URL segments that start with an underscore by prefixing the folder name with `%5F` (the URL-encoded form of an underscore): `%5FfolderName`.
- If you don't use private folders, it would be helpful to know Next.js [special file conventions](#) to prevent unexpected naming conflicts.

Route Groups

Route groups can be created by wrapping a folder in parenthesis: `(folderName)`

This indicates the folder is for organizational purposes and should **not be included** in the route's URL path.

Route groups are useful for:

- [Organizing routes into groups](#) e.g. by site section, intent, or team.
- Enabling nested layouts in the same route segment level:
 - [Creating multiple nested layouts in the same segment, including multiple root layouts](#)
 - [Adding a layout to a subset of routes in a common segment](#)

Next.js supports storing application code (including `app`) inside an optional `src` directory. This separates application code from project configuration files which mostly live in the root of a project.

Module Path Aliases

Next.js supports [Module Path Aliases](#) which make it easier to read and maintain imports across deeply nested project files.

`js app/dashboard/settings/analytics/page.js`

```
1 // before
2 import { Button } from '../../../../../components/button';
3
4 // after
5 import { Button } from '@/components/button';
```

Project organization strategies

There is no "right" or "wrong" way when it comes to organizing your own files and folders in Next.js a project.

The following section lists a very high-level overview of common strategies. The simplest takeaway is to choose a strategy that works for you and your team and be consistent across the project.

Note: In our examples below, we're using `components` and `lib` folders as generalized placeholders, their naming has

Store project files outside of app

This strategy stores all application code in shared folders in the **root of your project** and keeps the `app` directory purely for routing purposes.

Store project files in top-level folders inside of app

This strategy stores all application code in shared folders in the **root of the app directory**.

Split project files by feature or route

This strategy stores globally shared application code in the root `app` directory and **splits** more specific application code into the route segments that use them.

Next Steps

App Router > ... > Routing

Defining Routes

Learn how to create your first route in Next.js.

App Router > ... > Routing

Route Groups

Route Groups can be used to partition your Next.js application into different sections.

App Router > ... > Configuring

src Directory

Save pages under the `src` directory as an alternative to the root `pages` directory.

App Router > ... > Configuring

Absolute Imports and Module Path Aliases

Configure module path aliases that allow you to remap certain import paths.

> Menu

App Router > ... > Routing > Defining Routes

Defining Routes

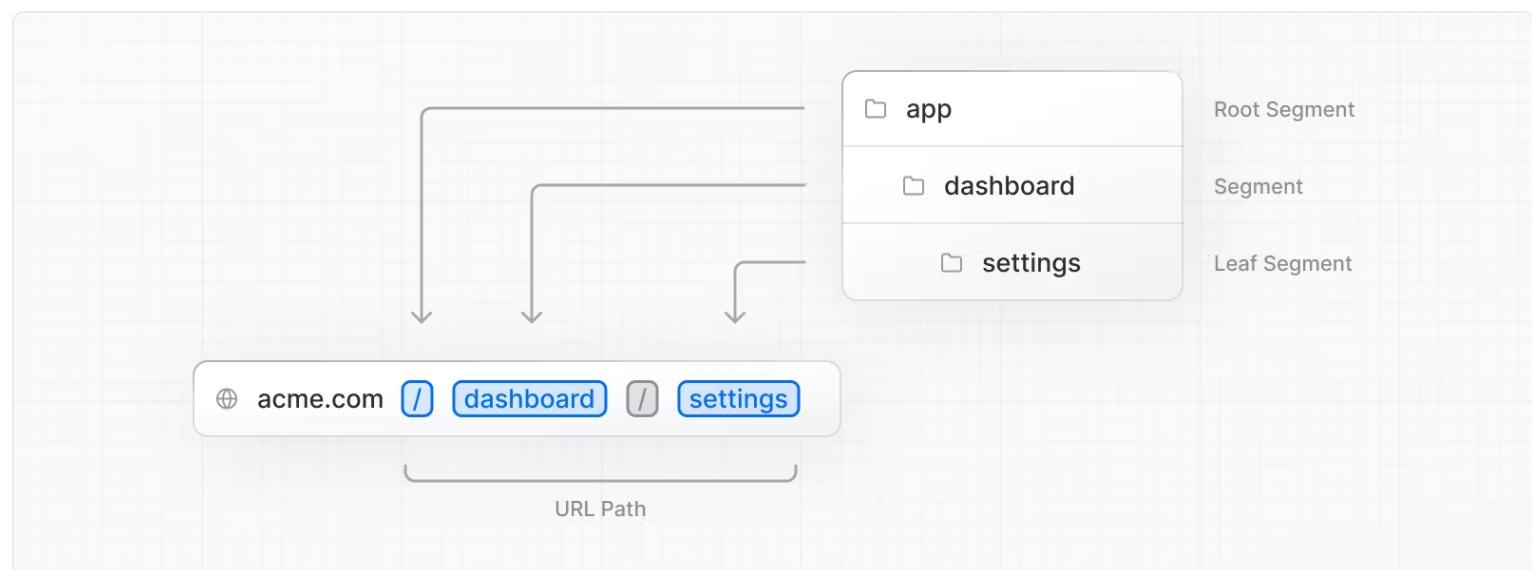
We recommend reading the [Routing Fundamentals](#) page before continuing.

This page will guide you through how to define and organize routes in your Next.js application.

Creating Routes

Next.js uses a file-system based router where **folders** are used to define routes.

Each folder represents a **route segment** that maps to a **URL** segment. To create a **nested route**, you can nest folders inside each other.



A special `page.js` file is used to make route segments publicly accessible.

In this example, the `/dashboard/analytics` URL path is *not* publicly accessible because it does not have a corresponding `page.js` file. This folder could be used to store components, stylesheets, images, or other colocated files.

Good to know: `.js`, `.jsx`, or `.tsx` file extensions can be used for special files.

Creating UI

[Special file conventions](#) are used to create UI for each route segment. The most common are [pages](#) to show UI unique to a route, and [layouts](#) to show UI that is shared across multiple routes.

For example, to create your first page, add a `page.js` file inside the `app` directory and export a React component:

app/page.tsx

```
1  export default function Page() {  
2    return <h1>Hello, Next.js!</h1>;  
3  }
```

Next Steps

Learn more about creating pages and layouts.

Pages and Layouts

Create your first page and shared layout with the App Router.

> Menu

App 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 are passed as the `params` prop to `layout`, `page`, `route`, and `generateMetadata` functions.

Example

For example, a blog could include the following route `app/blog/[slug]/page.js` where `[slug]` is the Dynamic Segment for blog posts.

JS app/blog/[slug]/page.js

```
1 export default function Page({ params }) {
2   return <div>My Post</div>;
3 }
```

Route	Example URL	params
-------	-------------	--------

app/blog/[slug]/page.js	/blog/a	{ slug: 'a' }
-------------------------	---------	---------------

app/blog/[slug]/page.js	/blog/b	{ slug: 'b' }
-------------------------	---------	---------------

Route	Example URL	params
app/blog/[slug]/page.js	/blog/c	{ slug: 'c' }

See the [generateStaticParams\(\)](#) page to learn how to generate the params for the segment.

Note: Dynamic Segments are equivalent to [Dynamic Routes](#) in the `pages` directory.

Generating Static Params

The `generateStaticParams` function can be used in combination with [dynamic route segments](#) to [statically generate](#) routes at build time instead of on-demand at request time.

TS app/blog/[slug]/page.tsx

```
1 export async function generateStaticParams() {
2   const posts = await fetch('https://.../posts').then((res) => res.json());
3
4   return posts.map((post) => ({
5     slug: post.slug,
6   }));
7 }
```

The primary benefit of the `generateStaticParams` function is its smart retrieval of data. If content is fetched within the `generateStaticParams` function using a `fetch` request, the requests are [automatically deduplicated](#). This means a `fetch` request with the same arguments across multiple `generateStaticParams`, Layouts, and Pages will only be made once, which decreases build times.

Use the [migration guide](#) if you are migrating from the `pages` directory.

See [generateStaticParams](#) server function documentation for more information and advanced use cases.

Catch-all Segments

Dynamic Segments can be extended to **catch-all** subsequent segments by adding an ellipsis inside the brackets `[...folderName]`.

For example, `app/shop/[...slug]/page.js` will match `/shop/clothes`, but also `/shop/clothes/tops`, `/shop/clothes/tops/t-shirts`, and so on.

Route	Example URL	params
<code>app/shop/[...slug]/page.js</code>	<code>/shop/a</code>	<code>{ slug: ['a'] }</code>
<code>app/shop/[...slug]/page.js</code>	<code>/shop/a/b</code>	<code>{ slug: ['a', 'b'] }</code>
<code>app/shop/[...slug]/page.js</code>	<code>/shop/a/b/c</code>	<code>{ slug: ['a', 'b', 'c'] }</code>

Optional Catch-all Segments

Catch-all Segments can be made **optional** by including the parameter in double square brackets:

`[[...folderName]]`.

For example, `app/shop/[[...slug]]/page.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
<code>app/shop/[[...slug]]/page.js</code>	<code>/shop</code>	<code>{}</code>
<code>app/shop/[[...slug]]/page.js</code>	<code>/shop/a</code>	<code>{ slug: ['a'] }</code>
<code>app/shop/[[...slug]]/page.js</code>	<code>/shop/a/b</code>	<code>{ slug: ['a', 'b'] }</code>
<code>app/shop/[[...slug]]/page.js</code>	<code>/shop/a/b/c</code>	<code>{ slug: ['a', 'b', 'c'] }</code>

TypeScript

When using TypeScript, you can add types for `params` depending on your configured route segment.

TS app/blog/[slug]/page.tsx

```
1 export default function Page({ params }: { params: { slug: string } }) {  
2     return <h1>My Page</h1>;
```

Route	params Type Definition
app/blog/[slug]/page.js	{ slug: string }
app/shop/[...slug]/page.js	{ slug: string[] }
app/[categoryId]/[itemId]/page.js	{ categoryId: string, itemId: string }

Note: This may be done automatically by the [TypeScript plugin](#) in the future.

Next Steps

For more information on what to do next, we recommend the following sections

App Router > ... > Routing

[Linking and Navigating](#)

Learn how navigation works in Next.js, and how to use the Link Component and `useRouter` hook.

App Router > ... > Functions

[generateStaticParams](#)

API reference for the generateStaticParams function.

> Menu

App Router > ... > Routing > Error Handling

Error Handling

The `error.js` file convention allows you to gracefully handle runtime errors in [nested routes](#).

- Automatically wrap a route segment and its nested children in a [React Error Boundary ↗](#).
- Create error UI tailored to specific segments using the file-system hierarchy to adjust granularity.
- Isolate errors to affected segments while keeping the rest of the app functional.
- Add functionality to attempt to recover from an error without a full page reload.

Create error UI by adding an `error.js` file inside a route segment and exporting a React component:

TS app/dashboard/error.tsx

▼



```
1  'use client'; // Error components must be Client Components
2
3  import { useEffect } from 'react';
4
5  export default function Error({
6    error,
7    reset,
8  }: {
9    error: Error;
10   
```

```
10   reset: () => void;
11 } {
12   useEffect(() => {
13     // Log the error to an error reporting service
14     console.error(error);
15   }, [error]);
16
17   return (
18     <div>
19       <h2>Something went wrong!</h2>
20       <button
21         onClick={
22           // Attempt to recover by trying to re-render the segment
23           () => reset()
24         }
25       >
26         Try again
27       </button>
28     </div>
29   );
30 }
```

How `error.js` Works

- `error.js` automatically creates an [React Error Boundary](#) that **wraps** a nested child segment or `page.js` component.
- The React component exported from the `error.js` file is used as the **fallback** component.

- If an error is thrown within the error boundary, the error is **contained**, and the fallback component is **rendered**.
- When the fallback error component is active, layouts **above** the error boundary **maintain** their state and **remain** interactive, and the error component can display functionality to recover from the error.

Recovering From Errors

The cause of an error can sometimes be temporary. In these cases, simply trying again might resolve the issue.

An error component can use the `reset()` function to prompt the user to attempt to recover from the error. When executed, the function will try to re-render the Error boundary's contents. If successful, the fallback error component is replaced with the result of the re-render.

app/dashboard/error.tsx

```
1  'use client';
2
3  export default function Error({
4    error,
5    reset,
6  }: {
7    error: Error;
8    reset: () => void;
9  }) {
10  return (
11    <div>
12      <h2>Something went wrong!</h2>
13      <button onClick={() => reset()}>Try again</button>
14    </div>
15  );
16}
```

Nested Routes

React components created through [special files](#) are rendered in a [specific nested hierarchy](#).

For example, a nested route with two segments that both include `layout.js` and `error.js` files are rendered in the following *simplified* component hierarchy:

The nested component hierarchy has implications for the behavior of `error.js` files across a nested route:

- Errors bubble up to the nearest parent error boundary. This means an `error.js` file will handle errors for all its nested child segments. More or less granular error UI can be achieved by placing `error.js` files at different levels in the nested folders of a route.
- An `error.js` boundary will **not** handle errors thrown in a `layout.js` component in the **same** segment because the error boundary is nested **inside** that layouts component.

Handling Errors in Layouts

`error.js` boundaries do **not** catch errors thrown in `layout.js` or `template.js` components of the same segment. This [intentional hierarchy](#) keeps important UI that is shared between sibling routes (such as navigation) visible and functional when an error occurs.

To handle errors within a specific layout or template, place an `error.js` file in the layouts parent segment.

To handle errors within the root layout or template, use a variation of `error.js` called `global-error.js`.

Handling Errors in Root Layouts

The root `app/error.js` boundary does **not** catch errors thrown in the root `app/layout.js` or `app/template.js` component.

To specifically handle errors in these root components, use a variation of `error.js` called `app/global-error.js` located in the root `app` directory.

Unlike the root `error.js`, the `global-error.js` error boundary wraps the **entire** application, and its fallback component replaces the root layout when active. Because of this, it is important to note that

`global-error.js` must define its own `<html>` and `<body>` tags.

`global-error.js` is the least granular error UI and can be considered "catch-all" error handling for the whole application. It is unlikely to be triggered often as root components are typically less dynamic, and other `error.js` boundaries will catch most errors.

Even if a `global-error.js` is defined, it is still recommended to define a root `error.js` whose fallback component will be rendered **within** the root layout, which includes globally shared UI and branding.

app/global-error.tsx

```
1  'use client';
2
3  export default function GlobalError({
4    error,
5    reset,
6  }: {
7    error: Error;
8    reset: () => void;
9  }) {
10  return (
11    <html>
12      <body>
13        <h2>Something went wrong!</h2>
14        <button onClick={() => reset()}>Try again</button>
15      </body>
16    </html>
17  );
18}
```

Handling Server Errors

If an error is thrown during `data fetching` or inside a Server Component, Next.js will forward the resulting `Error` object to the nearest `error.js` file as the `error` prop.

When running `next dev`, the `error` will be serialized and forwarded from the Server Component to the client `error.js`. To ensure security when running `next start` in production, a generic error message is forwarded to `error` along with a `.digest` which contains a hash of the error message. This hash can be used to correspond to server logs.

> Menu

App Router > ... > Routing > Intercepting Routes

Intercepting Routes

Intercepting routes allows you to load a route within the current layout while keeping the context for the current page. This routing paradigm can be useful when you want to "intercept" a certain route to show a different route.

For example, when clicking on a photo from within a feed, a modal overlaying the feed should show up with the photo. In this case, Next.js intercepts the `/feed` route and "masks" this URL to show `/photo/123` instead.

However, when navigating to the photo directly by for example when clicking a shareable URL or by refreshing the page, the entire photo page should render instead of the modal. No route interception should occur.

Convention

Intercepting routes can be defined with the `(..)` convention, which is similar to relative path convention `../` but for segments.

You can use:

- `(.)` to match segments on the **same level**
- `(..)` to match segments **one level above**
- `(..)(..)` to match segments **two levels above**
- `(...)` to match segments from the **root app directory**

For example, you can intercept the `photo` segment from within the `feed` segment by creating a `(...)photo` directory.

Note that the `(...)` convention is based on *route segments*, not the file-system.

Examples

Modals

Intercepting Routes can be used together with [Parallel Routes](#) to create modals.

Using this pattern to create modals overcomes some common challenges when working with modals, by allowing you to:

- Make the modal content **shareable through a URL**
- **Preserve context** when the page is refreshed, instead of closing the modal
- **Close the modal on backwards navigation** rather than going to the previous route
- **Reopen the modal on forwards navigation**

In the above example, the path to the `photo` segment can use the `(...)` matcher since `@modal` is a *slot* and not a *segment*. This means that the `photo` route is only one *segment* level higher, despite being two *file-system* levels higher.

Other examples could include opening a login modal in a top navbar while also having a dedicated [/login](#) page, or opening a shopping cart in a side modal.

[View an example ↗](#) of modals with Intercepted and Parallel Routes.

Next Steps

Learn how to use modals with Intercepted and Parallel Routes.

App Router > ... > Routing

Parallel Routes

Simultaneously render one or more pages in the same view that can be navigated independently. A pattern for highly dynamic applications.

> Menu

App Router > ... > Routing > Internationalization

Internationalization

Next.js enables you to configure the routing and rendering of content to support multiple languages. Making your site adaptive to different locales includes translated content (localization) and internationalized routes.

Terminology

- **Locale:** An identifier for a set of language and formatting preferences. This usually includes the preferred language of the user and possibly their geographic region.
 - `en-US`: English as spoken in the United States
 - `nl-NL`: Dutch as spoken in the Netherlands
 - `nl`: Dutch, no specific region

Routing Overview

It's recommended to use the user's language preferences in the browser to select which locale to use. Changing your preferred language will modify the incoming `Accept-Language` header to your application.

For example, using the following libraries, you can look at an incoming `Request` to determine which locale to select, based on the `Headers`, locales you plan to support, and the default locale.

JS middleware.js



```
1 import { match } from '@formatjs/intl-localematcher';
2 import Negotiator from 'negotiator';
3
4 let headers = { 'accept-language': 'en-US,en;q=0.5' };
```

```

5 let languages = new Negotiator({ headers }).languages();
6 let locales = ['en-US', 'nl-NL', 'nl'];
7 let defaultLocale = 'en-US';
8
9 match(languages, locales, defaultLocale); // -> 'en-US'

```

Routing can be internationalized by either the sub-path (`/fr/products`) or domain (`my-site.fr/products`). With this information, you can now redirect the user based on the locale inside [Middleware](#).

`js middleware.js`



```

1 import { NextResponse } from 'next/server'
2
3 let locales = ['en-US', 'nl-NL', 'nl']
4
5 // Get the preferred locale, similar to above or using a library
6 function getLocale(request) { ... }
7
8 export function middleware(request) {
9   // Check if there is any supported locale in the pathname
10  const pathname = request.nextUrl.pathname
11  const pathnameIsMissingLocale = locales.every(
12    (locale) => !pathname.startsWith(`/${locale}/`) && pathname !== `/${locale}`
13  )
14
15  // Redirect if there is no locale
16  if (pathnameIsMissingLocale) {
17    const locale = getLocale(request)
18
19    // e.g. incoming request is /products
20    // The new URL is now /en-US/products
21    return NextResponse.redirect(
22      new URL(`/${locale}/${pathname}`, request.url)
23    )
24  }
25 }
26
27 export const config = {
28   matcher: [
29     // Skip all internal paths (_next)
30     '/((?!_next).*)',
31     // Optional: only run on root (/) URL
32     // '/',
33   ],
34 }

```

Finally, ensure all special files inside `app/` are nested under `app/[lang]`. This enables the Next.js router to dynamically handle different locales in the route, and forward the `lang` parameter to every layout and page. For example:

```

1 // You now have access to the current locale
2 // e.g. /en-US/products -> `lang` is "en-US"
3 export default async function Page({ params: { lang } }) {
4   return ...
5 }
```

The root layout can also be nested in the new folder (e.g. `app/[lang]/layout.js`).

Localization

Changing displayed content based on the user's preferred locale, or localization, is not something specific to Next.js. The patterns described below would work the same with any web application.

Let's assume we want to support both English and Dutch content inside our application. We might maintain two different "dictionaries", which are objects that give us a mapping from some key to a localized string. For example:

```

1  {
2    "products": {
3      "cart": "Add to Cart"
4    }
5 }
```

```

1  {
2    "products": {
3      "cart": "Toevoegen aan Winkelwagen"
4    }
5 }
```

We can then create a `getDictionary` function to load the translations for the requested locale:

```

1 import 'server-only';
2
```

```
3 const dictionaries = {
4   en: () => import('./dictionaries/en.json').then((module) => module.default),
5   nl: () => import('./dictionaries/nl.json').then((module) => module.default),
6 };
7
8 export const getDictionary = async (locale) => dictionaries[locale]();
```

Given the currently selected language, we can fetch the dictionary inside of a layout or page.

js app/[lang]/page.js

```
1 import { getDictionary } from './dictionaries';
2
3 export default async function Page({ params: { lang } }) {
4   const dict = await getDictionary(lang); // en
5   return <button>{dict.products.cart}</button>; // Add to Cart
6 }
```

Because all layouts and pages in the `app/` directory default to [Server Components](#), we do not need to worry about the size of the translation files affecting our client-side JavaScript bundle size. This code will **only run on the server**, and only the resulting HTML will be sent to the browser.

Static Generation

To generate static routes for a given set of locales, we can use `generateStaticParams` with any page or layout. This can be global, for example, in the root layout:

js app/[lang]/layout.js

```
1 export async function generateStaticParams() {
2   return [{ lang: 'en-US' }, { lang: 'de' }];
3 }
4
5 export default function Root({ children, params }) {
6   return (
7     <html lang={params.lang}>
8       <body>{children}</body>
9     </html>
10   );
11 }
```

Examples

- [Minimal i18n routing and translations ↗](#)
- [next-intl ↗](#)

> Menu

App Router > ... > Routing > Linking and Navigating

Linking and Navigating

The Next.js router uses [server-centric routing](#) with [client-side navigation](#). It supports [instant loading states](#) and [concurrent rendering](#) ↗. This means navigation maintains client-side state, avoids expensive re-renders, is interruptible, and doesn't cause race conditions.

There are two ways to navigate between routes:

- [`<Link>` Component](#)
- [`useRouter` Hook](#)

This page will go through how to use `<Link>`, `useRouter()`, and dive deeper into how navigation works.

[`<Link>` Component](#)

`<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.

To use `<Link>`, import it from `next/link`, and pass a `href` prop to the component:

TS app/page.tsx

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

There are optional props you can pass to `<Link>`. See the [API reference](#) for more information.

Examples

Linking to Dynamic Segments

When linking to [dynamic segments](#), you can use [template literals and interpolation](#) ↗ to generate a list of links. For example, to generate a list of blog posts:

app/blog/PostList.jsx

```
1 import Link from 'next/link';
2
3 export default function PostList({ 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 }
```

Checking Active Links

You can use `usePathname()` to determine if a link is active. For example, to add a class to the active link, you can check if the current `pathname` matches the `href` of the link:

app/ui/Navigation.jsx

```
1 'use client';
2
3 import { usePathname } from 'next/navigation';
4 import { Link } from 'next/link';
5
6 export function Navigation({ navLinks }) {
7   const pathname = usePathname();
8
9   return (
10     <>
11       {navLinks.map((link) => {
12         const isActive = pathname.startsWith(link.href);
13
14         return (
15           <Link
16             className={isActive ? 'text-blue' : 'text-black'}
17             href={link.href}
18           >
```

```
18     key={link.name}
19     >
20         {link.name}
21     </Link>
22     );
23   )})
24   </>
25 );
26 }
```

Scrolling to an `id`

The default behavior of `<Link>` is to scroll to the top of the route segment that has changed. When there is an `id` defined in `href`, it will scroll to the specific `id`, similarly to a normal `<a>` tag.

To prevent scrolling to the top of the route segment, set `scroll={false}` and pass the add a hashed `id` to `href`:

```
1 <Link href="#/hashid" scroll={false}>
2   Scroll to specific id.
3 </Link>
```

useRouter() Hook

The `useRouter` hook allows you to programmatically change routes inside [Client Components](#).

To use `useRouter`, import it from `next/navigation`, and call the hook inside your Client Component:

app/page.jsx

```
1 'use client';
2
3 import { useRouter } from 'next/navigation';
4
5 export default function Page() {
6   const router = useRouter();
7
8   return (
9     <button type="button" onClick={() => router.push('/dashboard')}>
10       Dashboard
11     </button>
12   );
13 }
```

The `useRouter` provides methods such as `push()`, `refresh()`, and more. See the [API reference](#) for more information.

Recommendation: Use the `<Link>` component to navigate between routes unless you have a specific requirement for using `useRouter`.

How Navigation Works

- A route transition is initiated using `<Link>` or calling `router.push()`.
- The router updates the URL in the browser's address bar.
- The router avoids unnecessary work by re-using segments that haven't changed (e.g. shared layouts) from the [client-side cache](#). This is also referred to as [partial rendering](#).
- If the [conditions of soft navigation](#) are met, the router fetches the new segment from the cache rather than the server. If not, the router performs a [hard navigation](#) and fetches the Server Component payload from the server.
- If created, [loading UI](#) is shown from the server while the payload is being fetched.
- The router uses the cached or fresh payload to render the new segments on the client.

Client-side Caching of Rendered Server Components

Good to know: This client-side cache is different from the server-side [Next.js HTTP cache](#).

The new router has an **in-memory client-side cache** that stores the **rendered result** of Server Components (payload). The cache is split by route segments which allows invalidation at any level and ensures consistency across concurrent renders.

As users navigate around the app, the router will store the payload of previously fetched segments **and prefetched** segments in the cache.

This means, for certain cases, the router can re-use the cache instead of making a new request to the server. This improves performance by avoiding re-fetching data and re-rendering components unnecessarily.

Invalidating the Cache

[Server Actions](#) can be used to revalidate data on-demand by path (`revalidatePath`) or by cache tag (`revalidateTag`).

Prefetching

Prefetching is a way to preload a route in the background before it's visited. The rendered result of prefetched routes is added to the router's client-side cache. This makes navigating to a prefetched route near-instant.

By default, routes are prefetched as they become visible in the viewport when using the `<Link>` component. This can happen when the page first loads or through scrolling. Routes can also be programmatically prefetched using the `prefetch` method of the `useRouter()` hook.

Static and Dynamic Routes:

- If the route is static, all the Server Component payloads for the route segments will be prefetched.
- If the route is dynamic, the payload from the first shared layout down until the first `loading.js` file is prefetched. This reduces the cost of prefetching the whole route dynamically and allows [instant loading states](#) for dynamic routes.

Good to know:

- Prefetching is only enabled in production.
- Prefetching can be disabled by passing `prefetch={false}` to `<Link>`.

Soft Navigation

On navigation, the cache for changed segments is reused (if it exists), and no new requests are made to the server for data.

Conditions for Soft Navigation

On navigation, Next.js will use soft navigation if the route you are navigating to has been [prefetched](#), and either doesn't include [dynamic segments](#) or has the same dynamic parameters as the current route.

For example, consider the following route that includes a dynamic `[team]` segment:

`/dashboard/[team]/*`. The cached segments below `/dashboard/[team]/*` will only be invalidated when the `[team]` parameter changes.

- Navigating from `/dashboard/team-red/*` to `/dashboard/team-red/*` will be a soft navigation.
- Navigating from `/dashboard/team-red/*` to `/dashboard/team-blue/*` will be a hard navigation.

Hard Navigation

On navigation, the cache is invalidated and the server refetches data and re-renders the changed segments.

Back/Forward Navigation

Back and forward navigation ([popstate event ↗](#)) has a soft navigation behavior. This means, the client-side cache is re-used and navigation is near-instant.

Focus and Scroll Management

By default, Next.js will set focus and scroll into view the segment that's changed on navigation.

> Menu

App Router > ... > Routing > Loading UI and Streaming

Loading UI and Streaming

The special file `loading.js` helps you create meaningful Loading UI with [React Suspense ↗](#). With this convention, you can show an [instant loading state](#) from the server while the content of a route segment loads, the new content is automatically swapped in once rendering is complete.

Instant Loading States

An instant loading state is fallback UI that is shown immediately upon navigation. You can pre-render loading indicators such as skeletons and spinners, or a small but meaningful part of future screens such as a cover photo, title, etc. This helps users understand the app is responding and provides a better user experience.

Create a loading state by adding a `loading.js` file inside a folder.

TS app/dashboard/loading.tsx

```
1 export default function Loading() {  
2   // You can add any UI inside Loading, including a Skeleton.  
3   return <LoadingSkeleton />;  
4 }
```

In the same folder, `loading.js` will be nested inside `layout.js`. It will automatically wrap the `page.js` file and any children below in a `<Suspense>` boundary.

Good to know:

- Navigation is immediate, even with [server-centric routing](#).
- Navigation is interruptible, meaning changing routes does not need to wait for the content of the route to fully load before navigating to another route.

- Shared layouts remain interactive while new route segments load.

Recommendation: Use the `loading.js` convention for route segments (layouts and pages) as Next.js optimizes this functionality.

Streaming with Suspense

In addition to `loading.js`, you can also manually create Suspense Boundaries for your own UI components. The App Router supports streaming with [Suspense ↗](#) for both [Node.js](#) and [Edge runtimes](#).

What is Streaming?

To learn how Streaming works in React and Next.js, it's helpful to understand **Server-Side Rendering (SSR)** and its limitations.

With SSR, there's a series of steps that need to be completed before a user can see and interact with a page:

1. First, all data for a given page is fetched on the server.
2. The server then renders the HTML for the page.
3. The HTML, CSS, and JavaScript for the page are sent to the client.
4. A non-interactive user interface is shown using the generated HTML, and CSS.
5. Finally, React [hydrates ↗](#) the user interface to make it interactive.

These steps are sequential and blocking, meaning the server can only render the HTML for a page once all the data has been fetched. And, on the client, React can only hydrate the UI once the code for all

components in the page has been downloaded.

SSR with React and Next.js helps improve the perceived loading performance by showing a non-interactive page to the user as soon as possible.

However, it can still be slow as all data fetching on server needs to be completed before the page can be shown to the user.

Streaming allows you to break down the page's HTML into smaller chunks and progressively send those chunks from the server to the client.

This enables parts of the page to be displayed sooner, without waiting for all the data to load before any UI can be rendered.

Streaming works well with React's component model because each component can be considered a chunk. Components that have higher priority (e.g. product information) or that don't rely on data can be sent first (e.g. layout), and React can start hydration earlier. Components that have lower priority (e.g. reviews, related products) can be sent in the same server request after their data has been fetched.

Streaming is particularly beneficial when you want to prevent long data requests from blocking the page from rendering as it can reduce the [Time To First Byte \(TTFB\)](#) ↗ and [First Contentful Paint \(FCP\)](#) ↗. It also helps improve [Time to Interactive \(TTI\)](#) ↗, especially on slower devices.

Example

`<Suspense>` works by wrapping a component that performs an asynchronous action (e.g. fetch data), showing fallback UI (e.g. skeleton, spinner) while it's happening, and then swapping in your component once the action completes.

app/dashboard/page.tsx

```
1 import { Suspense } from 'react';
2 import { PostFeed, Weather } from './Components';
3
4 export default function Posts() {
5   return (
6     <section>
7       <Suspense fallback=<p>Loading feed...</p>>
8         <PostFeed />
9       </Suspense>
```

```
10     <Suspense fallback=<p>Loading weather...</p>>
11         <Weather />
12     </Suspense>
13     </section>
14   );
15 }
```

By using Suspense, you get the benefits of:

1. **Streaming Server Rendering** - Progressively rendering HTML from the server to the client.
2. **Selective Hydration** - React prioritizes what components to make interactive first based on user interaction.

For more Suspense examples and use cases, please see the [React Documentation ↗](#).

SEO

- Next.js will wait for data fetching inside `generateMetadata` to complete before streaming UI to the client. This guarantees the first part of a streamed response includes `<head>` tags.
- Since streaming is server-rendered, it does not impact SEO. You can use the [Mobile Friendly Test ↗](#) tool from Google to see how your page appears to Google's web crawlers and view the serialized HTML ([source ↗](#)).

> Menu

App 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` cookies. 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

App Router > ... > Routing > Pages and Layouts

Pages and Layouts

We recommend reading the [Routing Fundamentals](#) and [Defining Routes](#) pages before continuing.

The App Router inside Next.js 13 introduced new file conventions to easily create [pages](#), [shared layouts](#), and [templates](#). This page will guide you through how to use these special files in your Next.js application.

Pages

A page is UI that is **unique** to a route. You can define pages by exporting a component from a `page.js` file. Use nested folders to [define a route](#) and a `page.js` file to make the route publicly accessible.

Create your first page by adding a `page.js` file inside the `app` directory:

ts app/page.tsx

▼



```
1 // `app/page.tsx` is the UI for the `/` URL
2 export default function Page() {
3   return <h1>Hello, Home page!</h1>;
4 }
```

```
1 // `app/dashboard/page.tsx` is the UI for the `/dashboard` URL
2 export default function Page() {
3   return <h1>Hello, Dashboard Page!</h1>;
4 }
```

Good to know:

- A page is always the [leaf](#) of the [route subtree](#).
- [.js](#), [.jsx](#), or [.tsx](#) file extensions can be used for Pages.
- A [page.js](#) file is required to make a route segment publicly accessible.
- Pages are [Server Components](#) by default but can be set to a [Client Component](#).
- Pages can fetch data. View the [Data Fetching](#) section for more information.

Layouts

A layout is UI that is **shared** between multiple pages. On navigation, layouts preserve state, remain interactive, and do not re-render. Layouts can also be [nested](#).

You can define a layout by [default](#) exporting a React component from a [layout.js](#) file. The component should accept a [children](#) prop that will be populated with a child layout (if it exists) or a child page during rendering.

```
1 export default function DashboardLayout({
```

```
2   children, // will be a page or nested layout
3 }: {
4   children: React.ReactNode;
5 }) {
6   return (
7     <section>
8       /* Include shared UI here e.g. a header or sidebar */
9       <nav></nav>
10      {children}
11    </section>
12  );
13}
14 }
```

Good to know:

- The top-most layout is called the [Root Layout](#). This **required** layout is shared across all pages in an application. Root layouts must contain `html` and `body` tags.
- Any route segment can optionally define its own [Layout](#). These layouts will be shared across all pages in that segment.
- Layouts in a route are **nested** by default. Each parent layout wraps child layouts below it using the React `children` prop.
- You can use [Route Groups](#) to opt specific route segments in and out of shared layouts.
- Layouts are [Server Components](#) by default but can be set to a [Client Component](#).
- Layouts can fetch data. View the [Data Fetching](#) section for more information.
- Passing data between a parent layout and its children is not possible. However, you can fetch the same data in a route more than once, and React will [automatically dedupe the requests](#) without affecting performance.
- Layouts do not have access to the current route segment(s). To access route segments, you can use `useSelectedLayoutSegment` or `useSelectedLayoutSegments` in a Client Component.
- `.js`, `.jsx`, or `.tsx` file extensions can be used for Layouts.
- A `layout.js` and `page.js` file can be defined in the same folder. The layout will wrap the page.

Root Layout (Required)

The root layout is defined at the top level of the `app` directory and applies to all routes. This layout enables you to modify the initial HTML returned from the server.

TS app/layout.tsx

```
1 export default function RootLayout({
2   children,
3 }: {
4   children: React.ReactNode;
5 }) {
6   return (
7     <html lang="en">
8       <body>{children}</body>
```

```
9     </html>
10    );
11 }
```

Good to know:

- 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.
- You can use the [built-in SEO support](#) to manage `<head>` HTML elements, for example, the `<title>` element.
- You can use [route groups](#) to create multiple root layouts. See an [example here](#).
- The root layout is a [Server Component](#) by default and **can not** be set to a [Client Component](#).

Migrating from the `pages` directory: The root layout replaces the `_app.js` and `_document.js` files. [View the migration guide](#).

Nesting Layouts

Layouts defined inside a folder (e.g. `app/dashboard/layout.js`) apply to specific route segments (e.g. `acme.com/dashboard`) and render when those segments are active. By default, layouts in the file hierarchy are **nested**, which means they wrap child layouts via their `children` prop.

ts app/dashboard/layout.tsx

```
1 export default function DashboardLayout({
2   children,
3 }: {
4   children: React.ReactNode;
5 }) {
6   return <section>{children}</section>;
7 }
```

If you were to combine the two layouts above, the root layout (`app/layout.js`) would wrap the dashboard layout (`app/dashboard/layout.js`), which would wrap route segments inside `app/dashboard/*`.

The two layouts would be nested as such:

You can use [Route Groups](#) to opt specific route segments in and out of shared layouts.

Templates

Templates are similar to layouts in that they wrap each child layout or page. Unlike layouts that persist across routes and maintain state, templates create a new instance for each of their children on navigation. This means that when a user navigates between routes that share a template, a new instance of the component is mounted, DOM elements are recreated, state is **not** preserved, and effects are re-synchronized.

There may be cases where you need those specific behaviors, and templates would be a more suitable option than layouts. For example:

- Enter/exit animations using CSS or animation libraries.

- Features that rely on `useEffect` (e.g logging page views) and `useState` (e.g a per-page feedback form).
- To change the default framework behavior. For example, Suspense Boundaries inside layouts only show the fallback the first time the Layout is loaded and not when switching pages. For templates, the fallback is shown on each navigation.

Recommendation: We recommend using Layouts unless you have a specific reason to use Template.

A template can be defined by exporting a default React component from a `template.js` file. The component should accept a `children` prop which will be nested segments.

app/template.tsx

```
1 export default function Template({ children }: { children: React.ReactNode }) {
2   return <div>{children}</div>;
3 }
```

The rendered output of a route segment with a layout and a template will be as such:

Output

```
1 <Layout>
2   {/* Note that the template is given a unique key. */}
3   <Template key={routeParams}>{children}</Template>
4 </Layout>
```

Modifying `<head>`

In the `app` directory, you can modify the `<head>` HTML elements such as `title` and `meta` using the built-in SEO support.

Metadata can be defined by exporting a `metadata` object or `generateMetadata` function in a `layout.js` or `page.js` file.

TS app/page.tsx

```
1 import { Metadata } from 'next';
2
3 export const metadata: Metadata = {
4   title: 'Next.js',
5 };
6
7 export default function Page() {
8   return '...';
9 }
```

Good to know: You should **not** manually add `<head>` tags such as `<title>` and `<meta>` to root layouts. Instead, you should use the [Metadata API](#) which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.

Learn more about available metadata options in the [API reference](#).

> Menu

App Router > ... > Routing > Parallel Routes

Parallel Routes

Parallel Routing allows you to simultaneously or conditionally render one or more pages in the same layout. For highly dynamic sections of an app, such as dashboards and feeds on social sites, Parallel Routing can be used to implement complex routing patterns.

For example, you can simultaneously render the team and analytics pages.

Parallel Routing allows you to define independent error and loading states for each route as they're being streamed in independently.

Parallel Routing also allow you to conditionally render a slot based on certain conditions, such as authentication state. This enables fully separated code on the same URL.

Convention

Parallel routes are created using named **slots**. Slots are defined with the `@folder` convention, and are passed to the same-level layout as props.

Slots are *not* route segments and *do not affect the URL structure*. The file path `/@team/members` would be accessible at `/members`.

For example, the following file structure defines two explicit slots: `@analytics` and `@team`.

The folder structure above means that the component in `app/layout.js` now accepts the `@analytics` and `@team` slots props, and can render them in parallel alongside the `children` prop:

TS app/layout.tsx

```
1  export default function Layout(props: {
2    children: React.ReactNode;
3    analytics: React.ReactNode;
4    team: React.ReactNode;
5  }) {
6    return (
7      <>
8        {props.children}
9        {props.team}
10       {props.analytics}
11     </>
12   );
13 }
```

Good to know: The `children` prop is an implicit slot that does not need to be mapped to a folder. This means `app/page.js` is equivalent to `app/@children/page.js`.

Unmatched Routes

By default, the content rendered within a slot will match the current URL.

In the case of an unmatched slot, the content that Next.js renders differs based on the routing technique and folder structure.

default.js

You can define a `default.js` file to render as a fallback when Next.js cannot recover a slot's active state based on the current URL.

Consider the following folder structure. The `@team` slot has a `settings` directory, but `@analytics` does not.

If you were to navigate from the root `/` to `/settings`, the content that gets rendered is different based on the type of navigation and the availability of the `default.js` file.

	With <code>@analytics/default.js</code>	Without <code>@analytics/default.js</code>
Soft Navigation	<code>@team/settings/page.js</code> and <code>@analytics/page.js</code>	<code>@team/settings/page.js</code> and <code>@analytics/page.js</code>
Hard Navigation	<code>@team/settings/page.js</code> and <code>@analytics/default.js</code>	404

Soft Navigation

On a [soft navigation](#) - Next.js will render the slot's previously active state, even if it doesn't match the current URL.

Hard Navigation

On a [hard navigation](#) - a navigation that requires a full page reload - Next.js will first try to render the unmatched slot's `default.js` file. If that's not available, a 404 gets rendered.

The 404 for unmatched routes helps ensure that you don't accidentally render a route that shouldn't be parallel rendered.

useSelectedLayoutSegment(s)

Both `useSelectedLayoutSegment` and `useSelectedLayoutSegments` accept a `parallelRoutesKey`, which allows you read the active route segment within that slot.

app/layout.tsx

```
1 'use client';
2 import { useSelectedLayoutSegment } from 'next/navigation';
3
4 export default async function Layout(props: {
5   //...
6   authModal: React.ReactNode;
7 }) {
8   const loginSegments = useSelectedLayoutSegment('authModal');
9   // ...
10 }
```

When a user navigates to `@authModal/login`, or `/login` in the URL bar, `loginSegments` will be equal to the string `"login"`.

Examples

Modals

Parallel Routing can be used to render modals.

The `@authModal` slot renders a `<Modal>` component that can be shown by navigating to a matching route, for example `/login`.

TS app/layout.tsx

```
1 export default async function Layout(props: {
2   // ...
3   authModal: React.ReactNode;
4 }) {
5   return (
6     <>
7       {/* ... */}
8       {props.authModal}
9     </>
10   );
11 }
```

TS app/@authModal/login/page.tsx

```
1 import { Modal } from 'components/modal';
2
3 export default function Login() {
4   return (
5     <Modal>
6       <h1>Login</h1>
7       {/* ... */}
8     </Modal>
9   );
10 }
```

To ensure that the contents of the modal don't get rendered when it's not active, you can create a `default.js` file that returns `null`.

```
1 export default function Default() {  
2   return null;  
3 }
```

Dismissing a modal

If a modal was initiated through client navigation, e.g. by using `<Link href="/login">`, you can dismiss the modal by calling `router.back()` or by using a `Link` component.

```
1 'use client';  
2 import { useRouter } from 'next/navigation';  
3 import { Modal } from 'components/modal';  
4  
5 export default async function Login() {  
6   const router = useRouter();  
7   return (  
8     <Modal>  
9       <span onClick={() => router.back()}>Close modal</span>  
10      <h1>Login</h1>  
11      ...  
12    </Modal>  
13  );  
14}
```

More information on modals is covered in the [Intercepting Routes](#) section.

If you want to navigate elsewhere and dismiss a modal, you can also use a catch-all route.

JS app/@authModal/[...catchAll]/page.js



```
1 export default function CatchAll() {  
2   return null;  
3 }
```

Catch-all routes take precedence over `default.js`.

Conditional Routes

Parallel Routes can be used to implement conditional routing. For example, you can render a `@dashboard` or `@login` route depending on the authentication state.

TS app/layout.tsx



```
1 import { getUser } from '@/lib/auth';  
2  
3 export default function Layout({ params, dashboard, login }) {  
4   const isLoggedIn = getUser();  
5   return isLoggedIn ? dashboard : login;  
6 }
```


> Menu

App Router > ... > Routing > Route Groups

Route Groups

In the `app` directory, nested folders are normally mapped to URL paths. However, you can mark a folder as a **Route Group** to prevent the folder from being included in the route's URL path.

This allows you to organize your route segments and project files into logical groups without affecting the URL path structure.

Route groups are useful for:

- [Organizing routes into groups](#) e.g. by site section, intent, or team.
- Enabling [nested layouts](#) in the same route segment level:
 - [Creating multiple nested layouts in the same segment, including multiple root layouts](#)
 - [Adding a layout to a subset of routes in a common segment](#)

Convention

A route group can be created by wrapping a folder's name in parenthesis: `(folderName)`

Examples

Organize routes without affecting the URL path

To organize routes without affecting the URL, create a group to keep related routes together. The folders in parenthesis will be omitted from the URL (e.g. `(marketing)` or `(shop)`).

Even though routes inside `(marketing)` and `(shop)` share the same URL hierarchy, you can create a different layout for each group by adding a `layout.js` file inside their folders.

Opting specific segments into a layout

To opt specific routes into a layout, create a new route group (e.g. `(shop)`) and move the routes that share the same layout into the group (e.g. `account` and `cart`). The routes outside of the group will not share the layout (e.g. `checkout`).

Creating multiple root layouts

To create multiple [root layouts](#), remove the top-level `layout.js` file, and add a `layout.js` file inside each route groups. This is useful for partitioning an application into sections that have a completely different UI or experience. The `<html>` and `<body>` tags need to be added to each root layout.

In the example above, both `(marketing)` and `(shop)` have their own root layout.

Good to know:

- The naming of route groups has no special significance other than for organization. They do not affect the URL path.
- Routes that include a route group **should not** resolve to the same URL path as other routes. For example, since route groups don't affect URL structure, `(marketing)/about/page.js` and `(shop)/about/page.js` would both resolve to `/about` and cause an error.
- If you use multiple root layouts without a top-level `layout.js` file, your home `page.js` file should be defined in one of the route groups, For example: `app/(marketing)/page.js`.
- Navigating **across multiple root layouts** will cause a **full page load** (as opposed to a client-side navigation). For example, navigating from `/cart` that uses `app/(shop)/layout.js` to `/blog` that uses `app/(marketing)/layout.js` will cause a full page load. This **only** applies to multiple root layouts.

> Menu

App Router > ... > Routing > Route Handlers

Route Handlers

Route Handlers allow you to create custom request handlers for a given route using the [Web Request ↗](#) and [Response ↗](#) APIs.

Good to know: Route Handlers are only available inside the `app` directory. They are the equivalent of [API Routes](#) inside the `pages` directory meaning you **do not** need to use API Routes and Route Handlers together.

Convention

Route Handlers are defined in a `route.js|ts` file inside the `app` directory:

```
TS app/api/route.ts
```

```
export async function GET(request: Request) {}
```

Route Handlers can be nested inside the `app` directory, similar to `page.js` and `layout.js`. But there **cannot** be a `route.js` file at the same route segment level as `page.js`.

Supported HTTP Methods

The following [HTTP methods](#) are supported: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, and `OPTIONS`. If an unsupported method is called, Next.js will return a `405 Method Not Allowed` response.

Extended `NextRequest` and `NextResponse` APIs

In addition to supporting native [Request](#) and [Response](#). Next.js extends them with `NextRequest` and `NextResponse` to provide convenient helpers for advanced use cases.

Behavior

Static Route Handlers

Route Handlers are [statically evaluated](#) by default when using the `GET` method with the `Response` object.

app/items/route.ts

```
1 import { NextResponse } from 'next/server';
2
3 export async function GET() {
4   const res = await fetch('https://data.mongodb-api.com/...', {
5     headers: {
6       'Content-Type': 'application/json',
7       'API-Key': process.env.DATA_API_KEY,
8     },
9   });
10  const data = await res.json();
11
12  return NextResponse.json({ data });
13 }
```

TypeScript Warning: Although `Response.json()` is valid, native TypeScript types currently shows an error, you can use `NextResponse.json()` for typed responses instead.

Dynamic Route Handlers

Route handlers are evaluated dynamically when:

- Using the `Request` object with the `GET` method.
- Using any of the other HTTP methods.
- Using [Dynamic Functions](#) like `cookies` and `headers`.

- The [Segment Config Options](#) manually specifies dynamic mode.

For example:

TS app/products/api/route.ts

```
1 import { NextResponse } from 'next/server';
2
3 export async function GET(request: Request) {
4   const { searchParams } = new URL(request.url);
5   const id = searchParams.get('id');
6   const res = await fetch(`https://data.mongodb-api.com/product/${id}`, {
7     headers: {
8       'Content-Type': 'application/json',
9       'API-Key': process.env.DATA_API_KEY,
10    },
11  });
12  const product = await res.json();
13
14  return NextResponse.json({ product });
15 }
```

Similarly, the `POST` method will cause the Route Handler to be evaluated dynamically.

TS app/items/route.ts

```
1 import { NextResponse } from 'next/server';
2
3 export async function POST() {
4   const res = await fetch('https://data.mongodb-api.com/...', {
5     method: 'POST',
6     headers: {
7       'Content-Type': 'application/json',
8       'API-Key': process.env.DATA_API_KEY,
9     },
10    body: JSON.stringify({ time: new Date().toISOString() }),
11  });
12
13  const data = await res.json();
14
15  return NextResponse.json(data);
16 }
```

Note: Previously, API Routes could have been used for use cases like handling form submissions. Route Handlers are likely not the solution for these use cases. We will be recommending the use of [mutations](#) for this when ready.

Route Resolution

You can consider a `route` the lowest level routing primitive.

- They **do not** participate in layouts or client-side navigations like `page`.
- There **cannot** be a `route.js` file at the same route as `page.js`.

Page	Route	Result
<code>app/page.js</code>	<code>app/route.js</code>	Conflict
<code>app/page.js</code>	<code>app/api/route.js</code>	Valid
<code>app/[user]/page.js</code>	<code>app/api/route.js</code>	Valid

Each `route.js` or `page.js` file takes over all HTTP verbs for that route.

```
JS app/page.js 
```

```
1 export default function Page() {
2   return <h1>Hello, Next.js!</h1>;
3 }
4
5 // ✖ Conflict
6 // `app/route.js`
7 export async function POST(request) {}
```

Examples

The following examples show how to combine Route Handlers with other Next.js APIs and features.

Revalidating Static Data

You can **revalidate static data** fetches using the `next.revalidate` option:

```
TS app/items/route.ts 
```

```
1 import { NextResponse } from 'next/server';
2
3 export async function GET() {
4   const res = await fetch('https://data.mongodb-api.com/...', {
5     next: { revalidate: 60 }, // Revalidate every 60 seconds
6   });
7   const data = await res.json();
```

```
8
9     return NextResponse.json(data);
10 }
```

Alternatively, you can use the `revalidate` segment config option:

```
export const revalidate = 60;
```

Dynamic Functions

Route Handlers can be used with dynamic functions from Next.js, like `cookies` and `headers`.

Cookies

You can read cookies with `cookies` from `next/headers`. This server function can be called directly in a Route Handler, or nested inside of another function.

This `cookies` instance is read-only. To set cookies, you need to return a new `Response` using the `Set-Cookie` ↗ header.

app/api/route.ts

```
1 import { cookies } from 'next/headers';
2
3 export async function GET(request: Request) {
4     const cookieStore = cookies();
5     const token = cookieStore.get('token');
6
7     return new Response('Hello, Next.js!', {
8         status: 200,
9         headers: { 'Set-Cookie': `token=${token}` },
10    });
11 }
```

Alternatively, you can use abstractions on top of the underlying Web APIs to read cookies (`NextRequest`):

app/api/route.ts

```
1 import { type NextRequest } from 'next/server';
2
3 export async function GET(request: NextRequest) {
4     const token = request.cookies.get('token');
5 }
```

You can read headers with `headers` from `next/headers`. This server function can be called directly in a Route Handler, or nested inside of another function.

This `headers` instance is read-only. To set headers, you need to return a new `Response` with new `headers`.

app/api/route.ts

```
1 import { headers } from 'next/headers';
2
3 export async function GET(request: Request) {
4   const headersList = headers();
5   const referer = headersList.get('referer');
6
7   return new Response('Hello, Next.js!', {
8     status: 200,
9     headers: { referer: referer },
10    });
11 }
```

Alternatively, you can use abstractions on top of the underlying Web APIs to read headers (`NextRequest`):

app/api/route.ts

```
1 import { type NextRequest } from 'next/server';
2
3 export async function GET(request: NextRequest) {
4   const requestHeaders = new Headers(request.headers);
5 }
```

Redirects

app/api/route.ts

```
1 import { redirect } from 'next/navigation';
2
3 export async function GET(request: Request) {
4   redirect('https://nextjs.org/');
5 }
```

Dynamic Route Segments

We recommend reading the [Defining Routes](#) page before continuing.

Route Handlers can use [Dynamic Segments](#) to create request handlers from dynamic data.

JS app/items/[slug]/route.js



```
1 export async function GET(
2   request: Request,
3   {
4     params,
5   }: {
6     params: { slug: string };
7   },
8 ) {
9   const slug = params.slug; // 'a', 'b', or 'c'
10 }
```

Route	Example URL	params
app/items/[slug]/route.js	/items/a	{ slug: 'a' }
app/items/[slug]/route.js	/items/b	{ slug: 'b' }
app/items/[slug]/route.js	/items/c	{ slug: 'c' }

Streaming

TS app/api/route.ts



```
1 // https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream#convert_async_iterator
2 function iteratorToStream(iterator: any) {
3   return new ReadableStream({
4     async pull(controller) {
5       const { value, done } = await iterator.next();
6
7       if (done) {
8         controller.close();
9       } else {
10         controller.enqueue(value);
11       }
12     },
13   });
14 }
15
16 function sleep(time: number) {
17   return new Promise((resolve) => {
18     setTimeout(resolve, time);
19   });
20 }
21
22 const encoder = new TextEncoder();
```

```
24  async function* makeIterator() {
25    yield encoder.encode('<p>One</p>');
26    await sleep(200);
27    yield encoder.encode('<p>Two</p>');
28    await sleep(200);
29    yield encoder.encode('<p>Three</p>');
30  }
31
32 export async function GET() {
33   const iterator = makeIterator();
34   const stream = iteratorToStream(iterator);
35
36   return new Response(stream);
37 }
```

Request Body

You can read the `Request` body using the standard Web API methods:

TS app/items/route.ts

```
1 import { NextResponse } from 'next/server';
2
3 export async function POST(request: Request) {
4   const res = await request.json();
5   return NextResponse.json({ res });
6 }
```

CORS

You can set CORS headers on a `Response` using the standard Web API methods:

TS app/api/route.ts

```
1 export async function GET(request: Request) {
2   return new Response('Hello, Next.js!', {
3     status: 200,
4     headers: {
5       'Access-Control-Allow-Origin': '*',
6       'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
7       'Access-Control-Allow-Headers': 'Content-Type, Authorization',
8     },
9   });
10 }
```

Edge and Node.js Runtimes

Route Handlers have an isomorphic Web API to support both Edge and Node.js runtimes seamlessly, including support for streaming. Since Route Handlers use the same [route segment configuration](#) as Pages and Layouts, they support long-awaited features like general-purpose [statically regenerated](#) Route Handlers.

You can use the `runtime` segment config option to specify the runtime:

```
export const runtime = 'edge'; // 'nodejs' is the default
```

Non-UI Responses

You can use Route Handlers to return non-UI content. Note that [sitemap.xml](#), [robots.txt](#), [app icons](#), and [open graph images](#) all have built-in support.

TS app/rss.xml/route.ts

```
1 export async function GET() {
2   return new Response(`<?xml version="1.0" encoding="UTF-8" ?>
3 <rss version="2.0">
4
5 <channel>
6   <title>Next.js Documentation</title>
7   <link>https://nextjs.org/docs</link>
8   <description>The React Framework for the Web</description>
9 </channel>
10
11 </rss>`);
12 }
```

Segment Config Options

Route Handlers use the same [route segment configuration](#) as pages and layouts.

TS app/items/route.ts

```
1 export const dynamic = 'auto';
2 export const dynamicParams = true;
3 export const revalidate = false;
4 export const fetchCache = 'auto';
5 export const runtime = 'nodejs';
6 export const preferredRegion = 'auto';
```

See the [API reference](#) for more details.

API Reference

Learn more about the route.js file.

App Router > ... > File Conventions

route.js

API reference for the route.js special file.

> Menu

App 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 with CSS Modules.

[Tailwind CSS](#)

Style your Next.js Application using Tailwind CSS.

[CSS-in-JS](#)

Use CSS-in-JS libraries with Next.js

[Sass](#)

Style your Next.js application using Sass.

> Menu

App Router > ... > Styling > CSS-in-JS

CSS-in-JS

Warning: CSS-in-JS libraries which require runtime JavaScript are not currently supported in Server Components. Using CSS-in-JS with newer React features like Server Components and Streaming requires library authors to support the latest version of React, including [concurrent rendering ↗](#).

We're working with the React team on upstream APIs to handle CSS and JavaScript assets with support for React Server Components and streaming architecture.

The following libraries are supported in Client Components in the `app` directory:

- [styled-jsx](#)
- [styled-components](#)
- [tamagui](#) ↗
- [style9](#) ↗
- [vanilla-extract](#) ↗

The following are currently working on support:

- [emotion](#) ↗
- [Material UI](#) ↗
- [Chakra UI](#) ↗

Note: We're testing out different CSS-in-JS libraries and we'll be adding more examples for libraries that support React 18 features and/or the `app` directory.

If you want to style Server Components, we recommend using [CSS Modules](#) or other solutions that output CSS files, like PostCSS or [Tailwind CSS](#).

Configuring CSS-in-JS in `app`

Configuring CSS-in-JS is a three-step opt-in process that involves:

1. A **style registry** to collect all CSS rules in a render.
2. The new `useServerInsertedHTML` hook to inject rules before any content that might use them.
3. A Client Component that wraps your app with the style registry during initial server-side rendering.

styled-jsx

Using `styled-jsx` in Client Components requires using `v5.1.0`. First, create a new registry:

```
TS app/registry.tsx
```

```
1 'use client';
2
3 import React, { useState } from 'react';
4 import { useServerInsertedHTML } from 'next/navigation';
5 import { StyleRegistry, createStyleRegistry } from 'styled-jsx';
6
7 export default function StyledJsxRegistry({
8   children,
9 }: {
10   children: React.ReactNode;
11 }) {
12   // Only create stylesheet once with lazy initial state
13   // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
14   const [jsxStyleRegistry] = useState(() => createStyleRegistry());
15
16   useServerInsertedHTML(() => {
17     const styles = jsxStyleRegistry.styles();
18     jsxStyleRegistry.flush();
19     return <>{styles}</>;
20   });
21
22   return <StyleRegistry registry={jsxStyleRegistry}>{children}</StyleRegistry>;
23 }
```

Then, wrap your `root layout` with the registry:

```
TS app/layout.tsx
```

```
1 import StyledJsxRegistry from './registry';
2
3 export default function RootLayout({
4   children,
5 }: {
6   children: React.ReactNode;
7 }) {
8   return (
```

```
9   <html>
10     <body>
11       <StyledJsxRegistry>{children}</StyledJsxRegistry>
12     </body>
13   </html>
14 );
15 }
```

[View an example here ↗](#).

Styled Components

Below is an example of how to configure `styled-components@v6.0.0-rc.1` or greater:

First, use the `styled-components` API to create a global registry component to collect all CSS style rules generated during a render, and a function to return those rules. Then use the `useServerInsertedHTML` hook to inject the styles collected in the registry into the `<head>` HTML tag in the root layout.

```
TS lib/registry.tsx
```

```
1 'use client';
2
3 import React, { useState } from 'react';
4 import { useServerInsertedHTML } from 'next/navigation';
5 import { ServerStyleSheet, StyleSheetManager } from 'styled-components';
6
7 export default function StyledComponentsRegistry({
8   children,
9 }: {
10   children: React.ReactNode;
11 }) {
12   // Only create stylesheet once with lazy initial state
13   // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
14   const [styledComponentsStyleSheet] = useState(() => new ServerStyleSheet());
15
16   useServerInsertedHTML(() => {
17     const styles = styledComponentsStyleSheet.getStyleElement();
18     styledComponentsStyleSheet.instance.clearTag();
19     return <>{styles}</>;
20   });
21
22   if (typeof window !== 'undefined') return <>{children}</>;
23
24   return (
25     <StyleSheetManager sheet={styledComponentsStyleSheet.instance}>
26       {children}
27     </StyleSheetManager>
28   );
29 }
```

Wrap the `children` of the root layout with the style registry component:

app/layout.tsx

```
1 import StyledComponentsRegistry from './lib/registry';
2
3 export default function RootLayout({
4   children,
5 }: {
6   children: React.ReactNode;
7 }) {
8   return (
9     <html>
10       <body>
11         <StyledComponentsRegistry>{children}</StyledComponentsRegistry>
12       </body>
13     </html>
14   );
15 }
```

[View an example here ↗](#).

Good to know:

- During server rendering, styles will be extracted to a global registry and flushed to the `<head>` of your HTML. This ensures the style rules are placed before any content that might use them. In the future, we may use an upcoming React feature to determine where to inject the styles.
- During streaming, styles from each chunk will be collected and appended to existing styles. After client-side hydration is complete, `styled-components` will take over as usual and inject any further dynamic styles.
- We specifically use a Client Component at the top level of the tree for the style registry because it's more efficient to extract CSS rules this way. It avoids re-generating styles on subsequent server renders, and prevents them from being sent in the Server Component payload.

> Menu

App Router > ... > Styling > CSS Modules

CSS Modules

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

CSS Modules can be imported into any file inside the `app` directory:

TS app/dashboard/layout.tsx

```
1 import styles from './styles.module.css';
2
3 export default function DashboardLayout({
4   children,
5 }: {
6   children: React.ReactNode;
7 }) {
8   return <section className={styles.dashboard}>{children}</section>;
9 }
```

📄 app/dashboard/styles.module.css

```
1 .dashboard {
2   padding: 24px;
3 }
```

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

Global styles can be imported into any layout, page, or component inside the `app` directory.

Good to know: This is different from the `pages` directory, where you can only import global styles inside the `_app.js` file.

For example, consider a stylesheet named `app/global.css`:

```
1 body {  
2   padding: 20px 20px 60px;  
3   max-width: 680px;  
4   margin: 0 auto;  
5 }
```

Inside the root layout (`app/layout.js`), import the `global.css` stylesheet to apply the styles to every route in your application:

app/layout.tsx

```
1 // These styles apply to every route in the application  
2 import './global.css';  
3  
4 export default function RootLayout({  
5   children,  
6 }: {  
7   children: React.ReactNode;  
8 }) {  
9   return (  
10     <html lang="en">  
11       <body>{children}</body>  
12     </html>  
13   );  
14 }
```

External Stylesheets

Stylesheets published by external packages can be imported anywhere in the `app` directory, including colocated components:

```
TS app/layout.tsx ▾ ⌂

1 import 'bootstrap/dist/css/bootstrap.css';
2
3 export default function RootLayout({
4   children,
5 }: {
6   children: React.ReactNode;
7 }) {
8   return (
9     <html lang="en">
10       <body className="container">{children}</body>
11     </html>
12   );
13 }
```

Note: External stylesheets must be directly imported from an npm package or downloaded and colocated with your codebase. You cannot use `<link rel="stylesheet" />`.

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

App 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 }
```

📄 app/page.js

```
1 // maps to root `/` URL
2
3 import variables from './variables.module.scss';
4
5 export default function Page() {
6   return <h1 style={{ color: variables.primaryColor }}>Hello, Next.js!</h1>;
7 }
```

> Menu

App Router > ... > Styling > Tailwind CSS

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.

TS 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

App Router > Building Your Application > Upgrading

Upgrade Guide

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.

App Router Migration

Learn how to upgrade your existing Next.js application from the Pages Router to the App Router.

> Menu

App Router > ... > Upgrading > App Router Migration

App Router Incremental Adoption Guide

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`.

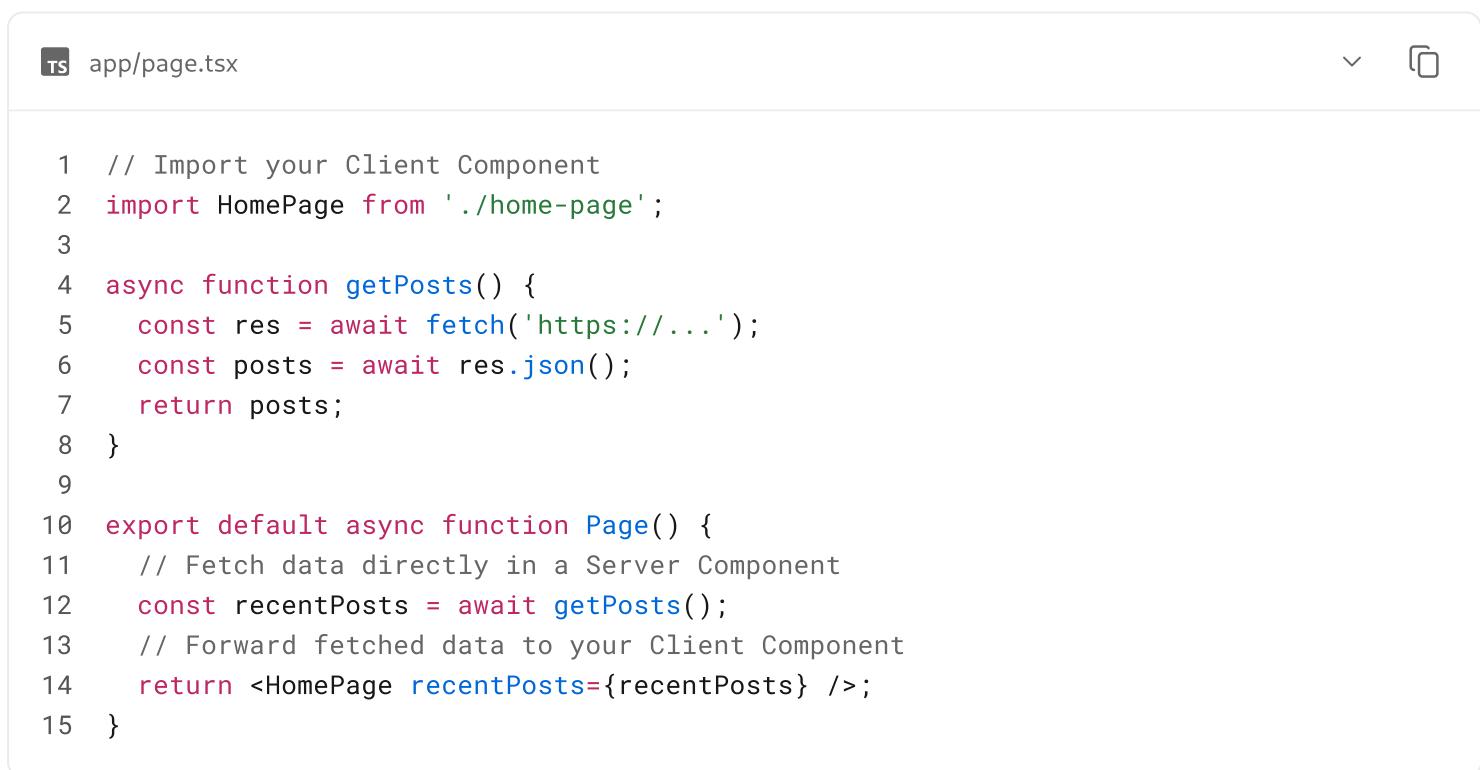
TS app/home-page.tsx

```
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.

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

App 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 ↗](#).