---
title: Installation
description: Create a new Next.js application with `create-next-app`. Set up TypeScript, styles, and configure your `next.config.js` file.
related:
  title: Next Steps
  description: Learn about the files and folders in your Next.js project.
  links:
    - getting-started/project-structure

System Requirements:

- [Node.js 18.17](https://nodejs.org/) or later.
- macOS, Windows (including WSL), and Linux are supported.

## Automatic Installation

We recommend starting a new Next.js app using [`create-next-app`](/docs/app/api-reference/create-next-app), which sets up everything automatically for you. To create a project, run:

```bash filename="Terminal"
npx create-next-app@latest
```

On installation, you'll see the following prompts:

```txt filename="Terminal"
What is your project named? my-app
Would you like to use TypeScript? No / Yes
Would you like to use ESLint? No / Yes
Would you like to use Tailwind CSS? No / Yes
Would you like to use `src/` directory? No / Yes
Would you like to use App Router? (recommended) No / Yes
Would you like to customize the default import alias (@/*)? No / Yes
What import alias would you like configured? @/*
```

After the prompts, `create-next-app` will create a folder with your project name and install the required dependencies.

> **Good to know**:
>
> - Next.js now ships with [TypeScript](/docs/app/building-your-application/configuring/typescript), [ESLint](/docs/app/building-your-application/configuring/eslint), and [Tailwind CSS](/docs/app/building-your-application/

styling/tailwind-css) configuration by default.
> - You can optionally use a [`src` directory](/docs/app/building-your-application/configuring/src-directory) in the root of your project to separate your application's code from configuration files.

## Manual Installation

To manually create a new Next.js app, install the required packages:

```bash filename="Terminal"
npm install next@latest react@latest react-dom@latest
```

Open your `package.json` file and add the following `scripts`:

```json filename="package.json"
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  }
}
```

These scripts refer to the different stages of developing an application:

- `dev`: runs [`next dev`](/docs/app/api-reference/next-cli#development) to start Next.js in development mode.
- `build`: runs [`next build`](/docs/app/api-reference/next-cli#build) to build the application for production usage.
- `start`: runs [`next start`](/docs/app/api-reference/next-cli#production) to start a Next.js production server.
- `lint`: runs [`next lint`](/docs/app/api-reference/next-cli#lint) to set up Next.js' built-in ESLint configuration.

### Creating directories

Next.js uses file-system routing, which means the routes in your application are determined by how you structure your files.

#### The `app` directory

For new applications, we recommend using the [App Router](/docs/app). This router allows you to use React's latest features and is an evolution of the [Pages Router](/docs/pages) based on community feedback.

Create an `app/` folder, then add a `layout.tsx` and `page.tsx` file. These will be rendered when the user visits the root of your application (`/`).

<Image
  alt="App Folder Structure"
  srcLight="/docs/light/app-getting-started.png"
  srcDark="/docs/dark/app-getting-started.png"
  width="1600"
  height="363"
/>

Create a [root layout](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required) inside `app/layout.tsx` with the required `<html>` and `<body>` tags:

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

Finally, create a home page `app/page.tsx` with some initial content:

```tsx filename="app/page.tsx" switcher
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```

```jsx filename="app/page.js" switcher
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```

> **Good to know**: If you forget to create `layout.tsx`, Next.js will automatically create this file when running the development server with `next dev`.

Learn more about [using the App Router](/docs/app/building-your-application/routing/defining-routes).

#### The `pages` directory (optional)

If you prefer to use the Pages Router instead of the App Router, you can create a `pages/` directory at the root of your project.

Then, add an `index.tsx` file inside your `pages` folder. This will be your home page (`/`):

```tsx filename="pages/index.tsx" switcher
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```

Next, add an `_app.tsx` file inside `pages/` to define the global layout. Learn more about the [custom App file](/docs/pages/building-your-application/routing/custom-app).

```tsx filename="pages/_app.tsx" switcher
import type { AppProps } from 'next/app'

export default function App({ Component, pageProps }: AppProps) {
  return <Component {...pageProps} />
}
```

```jsx filename="pages/_app.js" switcher
export default function App({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

Finally, add a `_document.tsx` file inside `pages/` to control the initial response from the server. Learn more about the [custom Document file](/docs/pages/building-your-application/routing/custom-document).

```tsx filename="pages/_document.tsx" switcher
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html>
      <Head />
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

Learn more about [using the Pages Router](/docs/pages/building-your-application/routing/pages-and-layouts).

> **Good to know**: Although you can use both routers in the same project, routes in `app` will be prioritized over `pages`. We recommend using only one router in your new project to avoid confusion.

#### The `public` folder (optional)

Create a `public` folder to store static assets such as images, fonts, etc. Files inside `public` directory can then be referenced by your code starting from the base URL (`/`).

## Run the Development Server

1. Run `npm run dev` to start the development server.
2. Visit `http://localhost:3000` to view your application.
3. Edit `app/page.tsx` (or `pages/index.tsx`) file and save it to see the updated result in your browser.

---
title: Next.js Project Structure
nav_title: Project Structure
description: A list of folders and files conventions in a Next.js project
---

This page provides an overview of the file and folder structure of a Next.js project. It covers top-level files and folders, configuration files, and routing conventions within the `app` and `pages` directories.

## Top-level folders

[`app`](/docs/app/building-your-application/routing)	App Router
[`pages`](/docs/pages/building-your-application/routing)	Pages Router
[`public`](/docs/app/building-your-application/optimizing/static-assets)	Static assets to be served
[`src`](/docs/app/building-your-application/configuring/src-directory)	Optional application source folder

## Top-level files

Next.js	
[`next.config.js`](/docs/app/api-reference/next-config-js)	Configuration file for Next.js
[`package.json`](/docs/getting-started/installation#manual-installation)	Project dependencies and scripts
[`instrumentation.ts`](/docs/app/building-your-application/optimizing/instrumentation)	OpenTelemetry and Instrumentation file
[`middleware.ts`](/docs/app/building-your-application/routing/middleware)	Next.js request middleware
[`.env`](/docs/app/building-your-application/configuring/environment-variables)	Environment variables
[`.env.local`](/docs/app/building-your-application/configuring/environment-variables)	Local environment variables
[`.env.production`](/docs/app/building-your-application/configuring/environment-variables)	Production environment variables
[`.env.development`](/docs/app/building-your-application/configuring/environment-variables)	Development environment variables
[`.eslintrc.json`](/docs/app/building-your-application/configuring/eslint)	Configuration file for ESLint
`.gitignore`	Git files and

folders to ignore |

`next-env.d.ts`	TypeScript declaration file for Next.js
`tsconfig.json`	Configuration file for TypeScript
`jsconfig.json`	Configuration file for JavaScript

## `app` Routing Conventions

### Routing Files

[`layout`](/docs/app/api-reference/file-conventions/layout)	`.js` `.jsx` `.tsx`	Layout
[`page`](/docs/app/api-reference/file-conventions/page)	`.js` `.jsx` `.tsx`	Page
[`loading`](/docs/app/api-reference/file-conventions/loading)	`.js` `.jsx` `.tsx`	Loading UI
[`not-found`](/docs/app/api-reference/file-conventions/not-found)	`.js` `.jsx` `.tsx`	Not found UI
[`error`](/docs/app/api-reference/file-conventions/error)	`.js` `.jsx` `.tsx`	Error UI
[`global-error`](/docs/app/api-reference/file-conventions/error#global-errorjs)	`.js` `.jsx` `.tsx`	Global error UI
[`route`](/docs/app/api-reference/file-conventions/route)	`.js` `.ts`	API endpoint
[`template`](/docs/app/api-reference/file-conventions/template)	`.js` `.jsx` `.tsx`	Re-rendered layout
[`default`](/docs/app/api-reference/file-conventions/default)	`.js` `.jsx` `.tsx`	Parallel route fallback page

### Nested Routes

[`folder`](/docs/app/building-your-application/routing#route-segments)	Route segment
[`folder/folder`](/docs/app/building-your-application/routing#nested-routes)	Nested route segment

### Dynamic Routes

[`[folder]`](/docs/app/building-your-application/routing/dynamic-routes#convention)	Dynamic route segment
[`[...folder]`](/docs/app/building-your-application/routing/dynamic-routes#catch-all-segments)	Catch-all route segment
[`[[...folder]]`](/docs/app/building-your-application/routing/dynamic-routes#optional-catch-all-segments)	Optional catch-all route segment

### Route Groups and Private Folders

[`(folder)`](/docs/app/building-your-application/routing/route-groups#convention)	Group routes without affecting routing
[`_folder`](/docs/app/building-your-application/routing/colocation#private-folders)	Opt folder and all child segments out of routing

### Parallel and Intercepted Routes

[`@folder`](/docs/app/building-your-application/routing/parallel-routes#convention)	Named slot
[`(.)folder`](/docs/app/building-your-application/routing/intercepting-routes#convention)	Intercept same level
[`(..)folder`](/docs/app/building-your-application/routing/intercepting-routes#convention)	Intercept one level above
[`(..)(..)folder`](/docs/app/building-your-application/routing/intercepting-routes#convention)	Intercept two levels above
[`(...)folder`](/docs/app/building-your-application/routing/intercepting-routes#convention)	Intercept from root

### Metadata File Conventions

#### App Icons

| |

[`favicon`](/docs/app/api-reference/file-conventions/metadata/app-icons#favicon)	`.ico`	Favicon file
[`icon`](/docs/app/api-reference/file-conventions/metadata/app-icons#icon)	`.ico` `.jpg` `.jpeg` `.png` `.svg`	App Icon file
[`icon`](/docs/app/api-reference/file-conventions/metadata/app-icons#generate-icons-using-code-js-ts-tsx)	`.js` `.ts` `.tsx`	Generated App Icon
[`apple-icon`](/docs/app/api-reference/file-conventions/metadata/app-icons#apple-icon)	`.jpg` `.jpeg`, `.png`	Apple App Icon file
[`apple-icon`](/docs/app/api-reference/file-conventions/metadata/app-icons#generate-icons-using-code-js-ts-tsx)	`.js` `.ts` `.tsx`	Generated Apple App Icon

#### Open Graph and Twitter Images

[`opengraph-image`](/docs/app/api-reference/file-conventions/metadata/opengraph-image#opengraph-image)	`.jpg` `.jpeg` `.png` `.gif`	Open Graph image file
[`opengraph-image`](/docs/app/api-reference/file-conventions/metadata/opengraph-image#generate-images-using-code-js-ts-tsx)	`.js` `.ts` `.tsx`	Generated Open Graph image
[`twitter-image`](/docs/app/api-reference/file-conventions/metadata/opengraph-image#twitter-image)	`.jpg` `.jpeg` `.png` `.gif`	Twitter image file
[`twitter-image`](/docs/app/api-reference/file-conventions/metadata/opengraph-image#generate-images-using-code-js-ts-tsx)	`.js` `.ts` `.tsx`	Generated Twitter image

#### SEO

[`sitemap`](/docs/app/api-reference/file-conventions/metadata/		

sitemap#static-sitemapxml) | `.xml` | Sitemap file |
[`sitemap`](/docs/app/api-reference/file-conventions/metadata/sitemap#generate-a-sitemap)	`.js` `.ts`	Generated Sitemap
[`robots`](/docs/app/api-reference/file-conventions/metadata/robots#static-robotstxt)	`.txt`	Robots file
[`robots`](/docs/app/api-reference/file-conventions/metadata/robots#generate-a-robots-file)	`.js` `.ts`	Generated Robots file

## `pages` Routing Conventions

### Special Files

[`_app`](/docs/pages/building-your-application/routing/custom-app)	`.js` `.jsx` `.tsx`	Custom App
[`_document`](/docs/pages/building-your-application/routing/custom-document)	`.js` `.jsx` `.tsx`	Custom Document
[`_error`](/docs/pages/building-your-application/routing/custom-error#more-advanced-error-page-customizing)	`.js` `.jsx` `.tsx`	Custom Error Page
[`404`](/docs/pages/building-your-application/routing/custom-error#404-page)	`.js` `.jsx` `.tsx`	404 Error Page
[`500`](/docs/pages/building-your-application/routing/custom-error#500-page)	`.js` `.jsx` `.tsx`	500 Error Page

### Routes

Folder convention		
[`index`](/docs/pages/building-your-application/routing/pages-and-layouts#index-routes)	`.js` `.jsx` `.tsx`	Home page
[`folder/index`](/docs/pages/building-your-application/routing/pages-and-layouts#index-routes)	`.js` `.jsx` `.tsx`	Nested page
File convention		
[`index`](/docs/pages/building-your-application/routing/pages-and-layouts#index-routes)	`.js` `.jsx` `.tsx`	Home page
[`file`](/docs/pages/building-your-application/routing/pages-and-layouts)	`.js` `.jsx` `.tsx`	Nested page

### Dynamic Routes

Folder convention		
[`[folder]/index`](/docs/pages/building-your-application/routing/dynamic-routes)	`.js` `.jsx` `.tsx`	Dynamic route segment
[`[...folder]/index`](/docs/pages/building-your-application/routing/dynamic-routes#catch-all-segments)	`.js` `.jsx` `.tsx`	Catch-all route segment
[`[[...folder]]/index`](/docs/pages/building-your-application/routing/dynamic-routes#optional-catch-all-segments)	`.js` `.jsx` `.tsx`	Optional catch-all route segment
File convention		
[`[file]`](/docs/pages/building-your-application/routing/dynamic-routes)	`.js` `.jsx` `.tsx`	Dynamic route segment
[`[...file]`](/docs/pages/building-your-application/routing/dynamic-routes#catch-all-segments)	`.js` `.jsx` `.tsx`	Catch-all route segment
[`[[...file]]`](/docs/pages/building-your-application/routing/dynamic-routes#optional-catch-all-segments)	`.js` `.jsx` `.tsx`	Optional catch-all route segment

---
title: Defining Routes
description: Learn how to create your first route in Next.js.
related:
  description: Learn more about creating pages and layouts.
  links:
    - app/building-your-application/routing/pages-and-layouts
---

> We recommend reading the [Routing Fundamentals](/docs/app/building-your-application/routing) 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](/docs/app/building-your-application/routing#route-segments) that maps to a **URL** segment. To create a [nested route](/docs/app/building-your-application/routing#nested-routes), you can nest folders inside each other.

<Image
  alt="Route segments to path segments"
  srcLight="/docs/light/route-segments-to-path-segments.png"
  srcDark="/docs/dark/route-segments-to-path-segments.png"
  width="1600"
  height="594"
/>

A special [`page.js` file](/docs/app/building-your-application/routing/pages-and-layouts#pages) is used to make route segments publicly accessible.

<Image
  alt="Defining Routes"
  srcLight="/docs/light/defining-routes.png"
  srcDark="/docs/dark/defining-routes.png"
  width="1600"
  height="687"
/>

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](/docs/app/building-your-application/routing#file-conventions) are used to create UI for each route segment. The most common are [pages](/docs/app/building-your-application/routing/pages-and-layouts#pages) to show UI unique to a route, and [layouts](/docs/app/building-your-application/routing/pages-and-layouts#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:

```tsx filename="app/page.tsx" switcher
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```

```jsx filename="app/page.js" switcher
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```

---
title: Pages and Layouts
description: Create your first page and shared layout with the App Router.

> We recommend reading the [Routing Fundamentals](/docs/app/building-your-application/routing) and [Defining Routes](/docs/app/building-your-application/routing/defining-routes) pages before continuing.

The App Router inside Next.js 13 introduced new file conventions to easily create [pages](#pages), [shared layouts](#layouts), and [templates](#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](/docs/app/building-your-application/routing/defining-routes) 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:

<Image
  alt="page.js special file"
  srcLight="/docs/light/page-special-file.png"
  srcDark="/docs/dark/page-special-file.png"
  width="1600"
  height="444"
/>

```tsx filename="app/page.tsx" switcher
// `app/page.tsx` is the UI for the `/` URL
export default function Page() {
```

  return <h1>Hello, Home page!</h1>
}

```jsx filename="app/page.js" switcher
// `app/page.js` is the UI for the `/` URL
export default function Page() {
  return <h1>Hello, Home page!</h1>
}
```

```tsx filename="app/dashboard/page.tsx" switcher
// `app/dashboard/page.tsx` is the UI for the `/dashboard` URL
export default function Page() {
  return <h1>Hello, Dashboard Page!</h1>
}
```

```jsx filename="app/dashboard/page.js" switcher
// `app/dashboard/page.js` is the UI for the `/dashboard` URL
export default function Page() {
  return <h1>Hello, Dashboard Page!</h1>
}
```

> **Good to know**:
>
> - A page is always the [leaf](/docs/app/building-your-application/routing#terminology) of the [route subtree](/docs/app/building-your-application/routing#terminology).
> - `.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](/docs/app/building-your-application/rendering/server-components) by default but can be set to a [Client Component](/docs/app/building-your-application/rendering/client-components).
> - Pages can fetch data. View the [Data Fetching](/docs/app/building-your-application/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](#nesting-layouts).

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.

<Image
  alt="layout.js special file"
  srcLight="/docs/light/layout-special-file.png"
  srcDark="/docs/dark/layout-special-file.png"
  width="1600"
  height="606"
/>

```tsx filename="app/dashboard/layout.tsx" switcher
export default function DashboardLayout({
  children, // will be a page or nested layout
}: {
  children: React.ReactNode
}) {
  return (
    <section>
      {/* Include shared UI here e.g. a header or sidebar */}
      <nav></nav>

      {children}
    </section>
  )
}
```

```jsx filename="app/dashboard/layout.js" switcher
export default function DashboardLayout({
  children, // will be a page or nested layout
}) {
  return (
    <section>
      {/* Include shared UI here e.g. a header or sidebar */}
      <nav></nav>

      {children}
    </section>
  )
}
```

> **Good to know**:
>
> - The top-most layout is called the [Root Layout](#root-layout-required). 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](#nesting-layouts). 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](/docs/app/building-your-application/routing/route-groups) to opt specific route segments in and out of shared layouts.
> - Layouts are [Server Components](/docs/app/building-your-application/rendering/server-components) by default but can be set to a [Client Component](/docs/app/building-your-application/rendering/client-components).
> - Layouts can fetch data. View the [Data Fetching](/docs/app/building-your-application/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](/docs/app/building-your-application/caching#request-memoization) without affecting performance.
> - Layouts do not have access to the route segments below itself. To access all route segments, you can use [`useSelectedLayoutSegment`](/docs/app/api-reference/functions/use-selected-layout-segment) or [`useSelectedLayoutSegments`](/docs/app/api-reference/functions/use-selected-layout-segments) 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.

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
export default function RootLayout({ children }) {
```

  return (
    <html lang="en">
      <body>{children}</body>
    </html>
)
}
```

> **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](/docs/app/building-your-application/optimizing/metadata) to manage `<head>` HTML elements, for example, the `<title>` element.
> - You can use [route groups](/docs/app/building-your-application/routing/route-groups) to create multiple root layouts. See an [example here](/docs/app/building-your-application/routing/route-groups#creating-multiple-root-layouts).
> - The root layout is a [Server Component](/docs/app/building-your-application/rendering/server-components) by default and **can not** be set to a [Client Component](/docs/app/building-your-application/rendering/client-components).

> **Migrating from the `pages` directory:** The root layout replaces the [`_app.js`](/docs/pages/building-your-application/routing/custom-app) and [`_document.js`](/docs/pages/building-your-application/routing/custom-document) files. [View the migration guide](/docs/app/building-your-application/upgrading/app-router-migration#migrating-_documentjs-and-_appjs).

### 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.

<Image
  alt="Nested Layout"
  srcLight="/docs/light/nested-layout.png"
  srcDark="/docs/dark/nested-layout.png"
  width="1600"
  height="606"
/>
```

```tsx filename="app/dashboard/layout.tsx" switcher
export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return <section>{children}</section>
}
```

```jsx filename="app/dashboard/layout.js" switcher
export default function DashboardLayout({ children }) {
  return <section>{children}</section>
}
```

> **Good to know**:
>
> - Only the root layout can contain `<html>` and `<body>` tags.

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:

<Image
  alt="Nested Layouts"
  srcLight="/docs/light/nested-layouts-ui.png"
  srcDark="/docs/dark/nested-layouts-ui.png"
  width="1600"
  height="1026"
/>

You can use [Route Groups](/docs/app/building-your-application/routing/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:

- 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.

A template can be defined by exporting a default React component from a `template.js` file. The component should accept a `children` prop.

<Image
  alt="template.js special file"
  srcLight="/docs/light/template-special-file.png"
  srcDark="/docs/dark/template-special-file.png"
  width="1600"
  height="444"
/>

```tsx filename="app/template.tsx" switcher
export default function Template({ children }: { children: React.ReactNode }) {
  return <div>{children}</div>
}
```

```jsx filename="app/template.js" switcher
export default function Template({ children }) {
  return <div>{children}</div>
}
```

In terms of nesting, `template.js` is rendered between a layout and its children. Here's a simplified output:

```jsx filename="Output"
<Layout>
  {/* Note that the template is given a unique key. */}
  <Template key={routeParam}>{children}</Template>
</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](/docs/app/building-your-application/optimizing/metadata).

Metadata can be defined by exporting a [`metadata` object](/docs/app/api-reference/functions/generate-metadata#the-metadata-object) or [`generateMetadata` function](/docs/app/api-reference/functions/generate-metadata#generatemetadata-function) in a [`layout.js`](/docs/app/api-reference/file-conventions/layout) or [`page.js`](/docs/app/api-reference/file-conventions/page) file.

```tsx filename="app/page.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'Next.js',
}

export default function Page() {
  return '...'
}
```

```jsx filename="app/page.js" switcher
export const metadata = {
  title: 'Next.js',
}

export default function Page() {
  return '...'
}
```

> **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](/docs/app/api-reference/functions/generate-metadata) which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.

[Learn more about available metadata options in the API reference.](/docs/app/api-reference/functions/generate-metadata)

---
title: Linking and Navigating
description: Learn how navigation works in Next.js, and how to use the Link Component and `useRouter` hook.
related:
  links:

  - app/building-your-application/caching
  - app/building-your-application/configuring/typescript

There are two ways to navigate between routes in Next.js:

- Using the [`<Link>` Component](#link-component)
- Using the [`useRouter` Hook](#userouter-hook)

This page will go through how to use `<Link>`, `useRouter()`, and dive deeper into how navigation works.

## `<Link>` Component

`<Link>` is a built-in component that extends the HTML `<a>` tag to provide [prefetching](#1-prefetching) and client-side navigation between routes. It is the primary way to navigate between routes in Next.js.

You can use it by importing it from `next/link`, and passing a `href` prop to the component:

```tsx filename="app/page.tsx" switcher
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

```jsx filename="app/page.js" switcher
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

There are other optional props you can pass to `<Link>`. See the [API reference](/docs/app/api-reference/components/link) for more.

### Examples

#### Linking to Dynamic Segments

When linking to [dynamic segments](/docs/app/building-your-application/routing/dynamic-routes), you can use [template literals and interpolation](https://developer.mozilla.org/docs/Web/JavaScript/Reference/

Template_literals) to generate a list of links. For example, to generate a list of blog posts:

```jsx filename="app/blog/PostList.js"
import Link from 'next/link'

export default function PostList({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blog/${post.slug}`}>{post.title}</Link>
        </li>
      ))}
    </ul>
  )
}
```

#### Checking Active Links

You can use [`usePathname()`](/docs/app/api-reference/functions/use-pathname) 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:

```tsx filename="app/components/links.tsx" switcher
'use client'

import { usePathname } from 'next/navigation'
import Link from 'next/link'

export function Links() {
  const pathname = usePathname()

  return (
    <nav>
      <ul>
        <li>
          <Link className={`link ${pathname === '/' ? 'active' : ''}`} href="/">
            Home
          </Link>
        </li>
        <li>
          <Link
            className={`link ${pathname === '/about' ? 'active' : ''}`}
            href="/about"
          >
            About
          </Link>
        </li>
      </ul>
    </nav>
  )
}
```

```jsx filename="app/components/links.js" switcher
'use client'

import { usePathname } from 'next/navigation'
import Link from 'next/link'

export function Links() {
  const pathname = usePathname()

  return (
    <nav>
      <ul>
        <li>
          <Link className={`link ${pathname === '/' ? 'active' : ''}`} href="/">
            Home
          </Link>
        </li>
        <li>
          <Link
            className={`link ${pathname === '/about' ? 'active' : ''}`}
            href="/about"
          >
            About
          </Link>
        </li>
      </ul>
    </nav>
  )
}
```

#### Scrolling to an `id`

The default behavior of the Next.js App Router is to scroll to the top of a new route or to maintain the scroll position for backwards and forwards navigation.

If you'd like to scroll to a specific `id` on navigation, you can append your URL with a `#` hash link or just pass a hash link to the `href` prop. This is possible since `<Link>` renders to an `<a>` element.

```jsx
<Link href="/dashboard#settings">Settings</Link>

// Output
<a href="/dashboard#settings">Settings</a>
```

#### Disabling scroll restoration

The default behavior of the Next.js App Router is to scroll to the top of a new route or to maintain the scroll position for backwards and forwards navigation. If you'd like to disable this behavior, you can pass `scroll={false}` to the `<Link>` component, or `scroll: false` to `router.push()` or `router.replace()`.

```jsx
// next/link
<Link href="/dashboard" scroll={false}>
  Dashboard
</Link>
```

```jsx
// useRouter
import { useRouter } from 'next/navigation'

const router = useRouter()

router.push('/dashboard', { scroll: false })
```

## `useRouter()` Hook

The `useRouter` hook allows you to programmatically change routes.

This hook can only be used inside Client Components and is imported from `next/navigation`.

```jsx filename="app/page.js"
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()
  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```

For a full list of `useRouter` methods, see the [API reference](/docs/app/api-reference/functions/use-router).

> **Recommendation:** Use the `<Link>` component to navigate between routes unless you have a specific requirement for using `useRouter`.

## How Routing and Navigation Works

The App Router uses a hybrid approach for routing and navigation. On the server, your application code is automatically code-split by route segments. And on the client, Next.js [prefetches](#1-prefetching) and [caches](#2-caching) the route segments. This means, when a user navigates to a new route, the browser doesn't reload the page, and only the route segments that change re-render - improving the navigation experience and performance.

### 1. Prefetching

Prefetching is a way to preload a route in the background before the user visits it.

There are two ways routes are prefetched in Next.js:

- **`<Link>` component**: Routes are automatically prefetched as they become visible in the user's viewport. Prefetching happens when the page first loads or when it comes into view through scrolling.
- **`router.prefetch()`**: The `useRouter` hook can be used to prefetch routes programmatically.

The`<Link>`'s prefetching behavior is different for static and dynamic routes:

- [**Static Routes**](/docs/app/building-your-application/rendering/server-components#static-rendering-default): `prefetch` defaults to `true`. The entire route is prefetched and cached.
- [**Dynamic Routes**](/docs/app/building-your-application/rendering/server-components#dynamic-rendering): `prefetch` default to automatic. Only the shared layout down until the first `loading.js` file is prefetched and cached for `30s`. This reduces the cost of fetching an entire dynamic route, and it means you can show an [instant loading state](/docs/app/building-your-application/

routing/loading-ui-and-streaming#instant-loading-states) for better visual feedback to users.

You can disable prefetching by setting the `prefetch` prop to `false`.

See the [`<Link>` API reference](/docs/app/api-reference/components/link) for more information.

> **Good to know**:
>
> - Prefetching is not enabled in development, only in production.

### 2. Caching

Next.js has an **in-memory client-side cache** called the [Router Cache](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#caching-data#router-cache). As users navigate around the app, the React Server Component Payload of [prefetched](#1-prefetching) route segments and visited routes are stored in the cache.

This means on navigation, the cache is reused as much as possible, instead of making a new request to the server - improving performance by reducing the number of requests and data transferred.

Learn more about how the [Router Cache](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#caching-data) works and how to configure it.

### 3. Partial Rendering

Partial rendering means only the route segments that change on navigation re-render on the client, and any shared segments are preserved.

For example, when navigating between two sibling routes, `/dashboard/settings` and `/dashboard/analytics`, the `settings` and `analytics` pages will be rendered, and the shared `dashboard` layout will be preserved.

<Image
  alt="How partial rendering works"
  srcLight="/docs/light/partial-rendering.png"
  srcDark="/docs/dark/partial-rendering.png"
  width="1600"
  height="945"
/>

Without partial rendering, each navigation would cause the full page to re-render on the server. Rendering only the segment that changes reduces the amount of data transferred and execution time, leading to improved performance.

### 4. Soft Navigation

By default, the browser performs a hard navigation between pages. This means the browser reloads the page and resets React state such as `useState` hooks in your app and browser state such as the user's scroll position or focused element. However, in Next.js, the App Router uses soft navigation. This means React only renders the segments that have changed while preserving React and browser state, and there is no full page reload.

### 5. Back and Forward Navigation

By default, Next.js will maintain the scroll position for backwards and forwards navigation, and re-use route segments in the [Router Cache](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#caching-data).

---
title: Route Groups
description: Route Groups can be used to partition your Next.js application into different sections.

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](#organize-routes-without-affecting-the-url-path) e.g. by site section, intent, or team.
- Enabling [nested layouts](/docs/app/building-your-application/routing/pages-and-layouts) in the same route segment level:
  - [Creating multiple nested layouts in the same segment, including multiple root layouts](#creating-multiple-root-layouts)
  - [Adding a layout to a subset of routes in a common segment](#opting-specific-segments-into-a-layout)

## 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)`).

<Image
  alt="Organizing Routes with Route Groups"
  srcLight="/docs/light/route-group-organisation.png"
  srcDark="/docs/dark/route-group-organisation.png"
  width="1600"
  height="930"
/>

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.

<Image
  alt="Route Groups with Multiple Layouts"
  srcLight="/docs/light/route-group-multiple-layouts.png"
  srcDark="/docs/dark/route-group-multiple-layouts.png"
  width="1600"
  height="768"
/>

### 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`).

<Image
  alt="Route Groups with Opt-in Layouts"
  srcLight="/docs/light/route-group-opt-in-layouts.png"
  srcDark="/docs/dark/route-group-opt-in-layouts.png"
  width="1600"
  height="930"
/>

### Creating multiple root layouts

To create multiple [root layouts](/docs/app/building-your-application/routing/

pages-and-layouts#root-layout-required), 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.

<Image
  alt="Route Groups with Multiple Root Layouts"
  srcLight="/docs/light/route-group-multiple-root-layouts.png"
  srcDark="/docs/dark/route-group-multiple-root-layouts.png"
  width="1600"
  height="687"
/>

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.

---
title: Dynamic Routes
description: Dynamic Routes can be used to programmatically generate route segments from dynamic data.
related:
  title: Next Steps
  description: For more information on what to do next, we recommend the following sections
  links:
    - app/building-your-application/routing/linking-and-navigating
    - app/api-reference/functions/generate-static-params

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](#generating-static-params) 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`](/docs/app/api-reference/file-conventions/layout), [`page`](/docs/app/api-reference/file-conventions/page), [`route`](/docs/app/building-your-application/routing/route-handlers), and [`generateMetadata`](/docs/app/api-reference/functions/generate-metadata#generatemetadata-function) 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.

```tsx filename="app/blog/[slug]/page.tsx" switcher
export default function Page({ params }: { params: { slug: string } }) {
  return <div>My Post: {params.slug}</div>
}
```

```jsx filename="app/blog/[slug]/page.js" switcher
export default function Page({ params }) {
  return <div>My Post: {params.slug}</div>
}
```

Route	Example URL	`params`
`app/blog/[slug]/page.js`	`/blog/a`	`{ slug: 'a' }`
`app/blog/[slug]/page.js`	`/blog/b`	`{ slug: 'b' }`
`app/blog/[slug]/page.js`	`/blog/c`	`{ slug: 'c' }`

See the [generateStaticParams()](#generating-static-params) page to learn how to generate the params for the segment.

> **Good to know**: Dynamic Segments are equivalent to [Dynamic Routes](/docs/app/building-your-application/routing/dynamic-routes) in the `pages` directory.

## Generating Static Params

The `generateStaticParams` function can be used in combination with [dynamic route segments](/docs/app/building-your-application/routing/dynamic-routes) to [**statically generate**](/docs/app/building-your-application/rendering/server-components#static-rendering-default) routes at build time instead of on-demand at request time.

```tsx filename="app/blog/[slug]/page.tsx" switcher
export async function generateStaticParams() {
  const posts = await fetch('https://.../posts').then((res) => res.json())

  return posts.map((post) => ({
    slug: post.slug,
  }))
}
```

```jsx filename="app/blog/[slug]/page.js" switcher
export async function generateStaticParams() {
  const posts = await fetch('https://.../posts').then((res) => res.json())

  return posts.map((post) => ({
    slug: post.slug,
  }))
}
```

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 memoized](/docs/app/building-your-application/caching#request-memoization). 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](/docs/app/building-your-application/upgrading/app-router-migration#dynamic-paths-getstaticpaths) if you are migrating from the `pages` directory.

See [`generateStaticParams` server function documentation](/docs/app/api-reference/functions/generate-static-params) 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`
`app/shop/[...slug]/page.js`	`/shop/a`	`{ slug: ['a'] }`
`app/shop/[...slug]/page.js`	`/shop/a/b`	`{ slug: ['a', 'b'] }`
`app/shop/[...slug]/page.js`	`/shop/a/b/c`	`{ slug: ['a', 'b', 'c'] }`

## Optional Catch-all Segments

Catch-all Segments can be made **optional** by including the parameter in double square brackets: `[[...folderName]]`.

For example, `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`
`app/shop/[[...slug]]/page.js`	`/shop`	`{}`
`app/shop/[[...slug]]/page.js`	`/shop/a`	`{ slug: ['a'] }`
`app/shop/[[...slug]]/page.js`	`/shop/a/b`	`{ slug: ['a', 'b'] }`
`app/shop/[[...slug]]/page.js`	`/shop/a/b/c`	`{ slug: ['a', 'b', 'c'] }`

## TypeScript

When using TypeScript, you can add types for `params` depending on your configured route segment.

```tsx filename="app/blog/[slug]/page.tsx" switcher
export default function Page({ params }: { params: { slug: string } }) {
  return <h1>My Page</h1>
}
```

```jsx filename="app/blog/[slug]/page.js" switcher
export default function Page({ params }) {
  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 }`

> **Good to know**: This may be done automatically by the [TypeScript plugin](/docs/app/building-your-application/configuring/typescript#typescript-plugin) in the future.

---
title: Loading UI and Streaming
description: 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.

The special file `loading.js` helps you create meaningful Loading UI with [React Suspense](https://react.dev/reference/react/Suspense). With this convention, you can show an [instant loading state](#instant-loading-states) from the server while the content of a route segment loads. The new content is automatically swapped in once rendering is complete.

<Image
  alt="Loading UI"
  srcLight="/docs/light/loading-ui.png"
  srcDark="/docs/dark/loading-ui.png"
  width="1600"
  height="691"
/>

## 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.

<Image
  alt="loading.js special file"
  srcLight="/docs/light/loading-special-file.png"
  srcDark="/docs/dark/loading-special-file.png"
  width="1600"
  height="606"
/>

```tsx filename="app/dashboard/loading.tsx" switcher
export default function Loading() {
  // You can add any UI inside Loading, including a Skeleton.
  return <LoadingSkeleton />
}
```

```jsx filename="app/dashboard/loading.js" switcher
export default function Loading() {
  // You can add any UI inside Loading, including a Skeleton.
  return <LoadingSkeleton />
}
```

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.

<Image
  alt="loading.js overview"
  srcLight="/docs/light/loading-overview.png"
  srcDark="/docs/dark/loading-overview.png"
  width="1600"
  height="768"
/>

> **Good to know**:
>
> - Navigation is immediate, even with [server-centric routing](/docs/app/building-your-application/routing/linking-and-navigating#how-routing-and-navigation-works).
> - 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](https://react.dev/reference/react/Suspense) for both [Node.js and Edge runtimes](/docs/app/building-your-application/rendering/edge-and-nodejs-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](https://react.dev/reference/react-dom/client/hydrateRoot#hydrating-server-rendered-html) the user interface to make it interactive.

<Image
  alt="Chart showing Server Rendering without Streaming"
  srcLight="/docs/light/server-rendering-without-streaming-chart.png"
  srcDark="/docs/dark/server-rendering-without-streaming-chart.png"
  width="1600"
  height="612"
/>

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.

<Image
  alt="Server Rendering without Streaming"
  srcLight="/docs/light/server-rendering-without-streaming.png"
  srcDark="/docs/dark/server-rendering-without-streaming.png"
  width="1600"
  height="748"
/>

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.

<Image
  alt="How Server Rendering with Streaming Works"
  srcLight="/docs/light/server-rendering-with-streaming.png"
  srcDark="/docs/dark/server-rendering-with-streaming.png"
  width="1600"
  height="785"
/>

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.

<Image
  alt="Chart showing Server Rendering with Streaming"
  srcLight="/docs/light/server-rendering-with-streaming-chart.png"
  srcDark="/docs/dark/server-rendering-with-streaming-chart.png"
  width="1600"
  height="730"
/>

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)](https://web.dev/ttfb/) and [First Contentful Paint (FCP)](https://web.dev/first-contentful-paint/). It also helps improve [Time to Interactive (TTI)](https://developer.chrome.com/en/docs/lighthouse/performance/interactive/), 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.

```tsx filename="app/dashboard/page.tsx" switcher
import { Suspense } from 'react'
import { PostFeed, Weather } from './Components'

export default function Posts() {
  return (
    <section>
```

```
      <Suspense fallback={<p>Loading feed...</p>}>
        <PostFeed />
      </Suspense>
      <Suspense fallback={<p>Loading weather...</p>}>
        <Weather />
      </Suspense>
    </section>
  )
}
```

```jsx filename="app/dashboard/page.js" switcher
import { Suspense } from 'react'
import { PostFeed, Weather } from './Components'

export default function Posts() {
  return (
    <section>
      <Suspense fallback={<p>Loading feed...</p>}>
        <PostFeed />
      </Suspense>
      <Suspense fallback={<p>Loading weather...</p>}>
        <Weather />
      </Suspense>
    </section>
  )
}
```

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](https://react.dev/reference/react/Suspense).

### SEO

- Next.js will wait for data fetching inside [`generateMetadata`](/docs/app/api-reference/functions/generate-metadata) 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](https://search.google.com/test/mobile-friendly) tool from

Google to see how your page appears to Google's web crawlers and view the serialized HTML ([source](https://web.dev/rendering-on-the-web/#seo-considerations)).

### Status Codes

When streaming, a `200` status code will be returned to signal that the request was successful.

The server can still communicate errors or issues to the client within the streamed content itself, for example, when using [`redirect`](/docs/app/api-reference/functions/redirect) or [`notFound`](/docs/app/api-reference/functions/not-found). Since the response headers have already been sent to the client, the status code of the response cannot be updated. This does not affect SEO.

---
title: Error Handling
description: Handle runtime errors by automatically wrapping route segments and their nested children in a React Error Boundary.
related:
  links:
    - app/api-reference/file-conventions/error

The `error.js` file convention allows you to gracefully handle unexpected runtime errors in [nested routes](/docs/app/building-your-application/routing#nested-routes).

- Automatically wrap a route segment and its nested children in a [React Error Boundary](https://react.dev/reference/react/Component#catching-rendering-errors-with-an-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 application 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:

<Image
  alt="error.js special file"
  srcLight="/docs/light/error-special-file.png"
  srcDark="/docs/dark/error-special-file.png"
  width="1600"

  height="606"
/>

```tsx filename="app/dashboard/error.tsx" switcher
'use client' // Error components must be Client Components

import { useEffect } from 'react'

export default function Error({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  useEffect(() => {
    // Log the error to an error reporting service
    console.error(error)
  }, [error])

  return (
    <div>
      <h2>Something went wrong!</h2>
      <button
        onClick={
          // Attempt to recover by trying to re-render the segment
          () => reset()
        }
      >
        Try again
      </button>
    </div>
  )
}
```

```jsx filename="app/dashboard/error.js" switcher
'use client' // Error components must be Client Components

import { useEffect } from 'react'

export default function Error({ error, reset }) {
  useEffect(() => {
    // Log the error to an error reporting service
    console.error(error)
  }, [error])
```

  return (
    <div>
      <h2>Something went wrong!</h2>
      <button
        onClick={
          // Attempt to recover by trying to re-render the segment
          () => reset()
        }
      >
        Try again
      </button>
    </div>
)
}
```

### How `error.js` Works

<Image
  alt="How error.js works"
  srcLight="/docs/light/error-overview.png"
  srcDark="/docs/dark/error-overview.png"
  width="1600"
  height="903"
/>

- `error.js` automatically creates a [React Error Boundary](https://react.dev/reference/react/Component#catching-rendering-errors-with-an-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.
```

```tsx filename="app/dashboard/error.tsx" switcher
'use client'

export default function Error({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  return (
    <div>
      <h2>Something went wrong!</h2>
      <button onClick={() => reset()}>Try again</button>
    </div>
  )
}
```

```jsx filename="app/dashboard/error.js" switcher
'use client'

export default function Error({ error, reset }) {
  return (
    <div>
      <h2>Something went wrong!</h2>
      <button onClick={() => reset()}>Try again</button>
    </div>
  )
}
```

### Nested Routes

React components created through [special files](/docs/app/building-your-application/routing#file-conventions) are rendered in a [specific nested hierarchy](/docs/app/building-your-application/routing#component-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:

<Image
  alt="Nested Error Component Hierarchy"
  srcLight="/docs/light/nested-error-component-hierarchy.png"
  srcDark="/docs/dark/nested-error-component-hierarchy.png"

  width="1600"
  height="687"
/>

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 layout's 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](#nested-routes) 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 layout's 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.

```tsx filename="app/global-error.tsx" switcher
'use client'

export default function GlobalError({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  return (
    <html>
      <body>
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
      </body>
    </html>
  )
}
```

```jsx filename="app/global-error.js" switcher
'use client'

export default function GlobalError({ error, reset }) {
  return (
    <html>
      <body>
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
      </body>
    </html>
  )
}
```

### Handling Server Errors

If an error is thrown inside a Server Component, Next.js will forward an `Error` object (stripped of sensitive error information in production) to the nearest `error.js` file as the `error` prop.

#### Securing Sensitive Error Information

During production, the `Error` object forwarded to the client only includes a generic `message` and `digest` property.

This is a security precaution to avoid leaking potentially sensitive details included in the error to the client.

The `message` property contains a generic message about the error and the `digest` property contains an automatically generated hash of the error that can be used to match the corresponding error in server-side logs.

During development, the `Error` object forwarded to the client will be serialized and include the `message` of the original error for easier debugging.

---
title: Parallel Routes
description: Simultaneously render one or more pages in the same view that can be navigated independently. A pattern for highly dynamic applications.

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.

```
<Image
  alt="Parallel Routes Diagram"
  srcLight="/docs/light/parallel-routes.png"
  srcDark="/docs/dark/parallel-routes.png"
  width="1600"
  height="952"
/>
```

Parallel Routing allows you to define independent error and loading states for each route as they're being streamed in independently.

```
<Image
  alt="Parallel routes enable custom error and loading states"
  srcLight="/docs/light/parallel-routes-cinematic-universe.png"
  srcDark="/docs/dark/parallel-routes-cinematic-universe.png"
  width="1600"
  height="1218"
/>
```

Parallel Routing also allows you to conditionally render a slot based on certain conditions, such as authentication state. This enables fully separated code on the same URL.

<Image
  alt="Conditional routes diagram"
  srcLight="/docs/light/conditional-routes-ui.png"
  srcDark="/docs/dark/conditional-routes-ui.png"
  width="1600"
  height="898"
/>

## 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`.

<Image
  alt="Parallel Routes File-system Structure"
  srcLight="/docs/light/parallel-routes-file-system.png"
  srcDark="/docs/dark/parallel-routes-file-system.png"
  width="1600"
  height="687"
/>

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:

```tsx filename="app/layout.tsx" switcher
export default function Layout(props: {
  children: React.ReactNode
  analytics: React.ReactNode
  team: React.ReactNode
}) {
  return (
    <>
      {props.children}
      {props.team}
      {props.analytics}
    </>
  )
```

}

```jsx filename="app/layout.js" switcher
export default function Layout(props) {
  return (
    <>
      {props.children}
      {props.team}
      {props.analytics}
    </>
  )
}
```

> **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.

<Image
  alt="Parallel Routes unmatched routes"
  srcLight="/docs/light/parallel-routes-unmatched-routes.png"
  srcDark="/docs/dark/parallel-routes-unmatched-routes.png"
  width="1600"
  height="930"
/>

#### Navigation

On navigation, Next.js will render the slot's previously active state, even if it doesn't match the current URL.

#### Reload

On 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`](/docs/app/api-reference/functions/use-selected-layout-segment) and [`useSelectedLayoutSegments`](/docs/app/api-reference/functions/use-selected-layout-segments) accept a `parallelRoutesKey`, which allows you to read the active route segment within that slot.

```tsx filename="app/layout.tsx" switcher
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

export default async function Layout(props: {
  //...
  auth: React.ReactNode
}) {
  const loginSegments = useSelectedLayoutSegment('auth')
  // ...
}
```

```jsx filename="app/layout.js" switcher
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

export default async function Layout(props) {
  const loginSegments = useSelectedLayoutSegment('auth')
  // ...
}
```

When a user navigates to `@auth/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.

<Image
  alt="Parallel Routes Diagram"
  srcLight="/docs/light/parallel-routes-auth-modal.png"
  srcDark="/docs/dark/parallel-routes-auth-modal.png"
  width="1600"
  height="687"
/>

The `@auth` slot renders a `<Modal>` component that can be shown by navigating to a matching route, for example `/login`.

```tsx filename="app/layout.tsx" switcher
export default async function Layout(props: {
  // ...
  auth: React.ReactNode
}) {
  return (
    <>
      {/* ... */}
      {props.auth}
    </>
  )
}
```

```jsx filename="app/layout.js" switcher
export default async function Layout(props) {
  return (
    <>
      {/* ... */}
      {props.auth}
    </>
  )
}
```

```tsx filename="app/@auth/login/page.tsx" switcher
import { Modal } from 'components/modal'

export default function Login() {
  return (
    <Modal>
```

      <h1>Login</h1>
      {/* ... */}
    </Modal>
)
}

```jsx filename="app/@auth/login/page.js" switcher
import { Modal } from 'components/modal'

export default function Login() {
  return (
    <Modal>
      <h1>Login</h1>
      {/* ... */}
    </Modal>
  )
}
```

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`.

```tsx filename="app/@auth/default.tsx" switcher
export default function Default() {
  return null
}
```

```jsx filename="app/@auth/default.js" switcher
export default function Default() {
  return null
}
```

#### 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.

```tsx filename="app/@auth/login/page.tsx" highlight="5" switcher
'use client'
import { useRouter } from 'next/navigation'
import { Modal } from 'components/modal'

export default async function Login() {
```

  const router = useRouter()
  return (
    <Modal>
      <span onClick={() => router.back()}>Close modal</span>
      <h1>Login</h1>
      ...
    </Modal>
)
}

```jsx filename="app/@auth/login/page.js" highlight="5" switcher
'use client'
import { useRouter } from 'next/navigation'
import { Modal } from 'components/modal'

export default async function Login() {
  const router = useRouter()
  return (
    <Modal>
      <span onClick={() => router.back()}>Close modal</span>
      <h1>Login</h1>
      ...
    </Modal>
  )
}
```

> More information on modals is covered in the [Intercepting Routes](/docs/app/building-your-application/routing/intercepting-routes) section.

If you want to navigate elsewhere and dismiss a modal, you can also use a catch-all route.

<Image
  alt="Parallel Routes Diagram"
  srcLight="/docs/light/parallel-routes-catchall.png"
  srcDark="/docs/dark/parallel-routes-catchall.png"
  width="1600"
  height="768"
/>

```tsx filename="app/@auth/[...catchAll]/page.tsx" switcher
export default function CatchAll() {
  return null
}
```

```jsx filename="app/@auth/[...catchAll]/page.js" switcher
export default function CatchAll() {
  return null
}
```

> 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.

```tsx filename="app/layout.tsx" switcher
import { getUser } from '@/lib/auth'

export default function Layout({
  dashboard,
  login,
}: {
  dashboard: React.ReactNode
  login: React.ReactNode
}) {
  const isLoggedIn = getUser()
  return isLoggedIn ? dashboard : login
}
```

```jsx filename="app/layout.js" switcher
import { getUser } from '@/lib/auth'

export default function Layout({ dashboard, login }) {
  const isLoggedIn = getUser()
  return isLoggedIn ? dashboard : login
}
```

<Image
  alt="Parallel routes authentication example"
  srcLight="/docs/light/conditional-routes-ui.png"
  srcDark="/docs/dark/conditional-routes-ui.png"
  width="1600"
  height="898"
/>

---
title: Intercepting Routes
description: 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.
related:
  title: Next Steps
  description: Learn how to use modals with Intercepted and Parallel Routes.
  links:
    - app/building-your-application/routing/parallel-routes

Intercepting routes allows you to load a route from another part of your application within the current layout. This routing paradigm can be useful when you want to display the content of a route without the user switching to a different context.

For example, when clicking on a photo in a feed, you can display the photo in a modal, overlaying the feed. In this case, Next.js intercepts the `/photo/123` route, masks the URL, and overlays it over `/feed`.

<Image
  alt="Intercepting routes soft navigation"
  srcLight="/docs/light/intercepting-routes-soft-navigate.png"
  srcDark="/docs/dark/intercepting-routes-soft-navigate.png"
  width="1600"
  height="617"
/>

However, when navigating to the photo by clicking a shareable URL or by refreshing the page, the entire photo page should render instead of the modal. No route interception should occur.

<Image
  alt="Intercepting routes hard navigation"
  srcLight="/docs/light/intercepting-routes-hard-navigate.png"
  srcDark="/docs/dark/intercepting-routes-hard-navigate.png"
  width="1600"
  height="604"
/>

## 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.

<Image
  alt="Intercepting routes folder structure"
  srcLight="/docs/light/intercepted-routes-files.png"
  srcDark="/docs/dark/intercepted-routes-files.png"
  width="1600"
  height="604"
/>

> Note that the `(..)` convention is based on _route segments_, not the file-system.

## Examples

### Modals

Intercepting Routes can be used together with [Parallel Routes](/docs/app/building-your-application/routing/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**

<Image
  alt="Intercepting routes modal example"
  srcLight="/docs/light/intercepted-routes-modal-example.png"
  srcDark="/docs/dark/intercepted-routes-modal-example.png"
  width="1600"
  height="976"
/>

> 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](https://github.com/vercel-labs/nextgram) of modals with Intercepted and Parallel Routes.

---
title: Route Handlers
description: Create custom request handlers for a given route using the Web's Request and Response APIs.
related:
  title: API Reference
  description: Learn more about the route.js file.
  links:
    - app/api-reference/file-conventions/route

Route Handlers allow you to create custom request handlers for a given route using the Web [Request](https://developer.mozilla.org/docs/Web/API/Request) and [Response](https://developer.mozilla.org/docs/Web/API/Response) APIs.

<Image
  alt="Route.js Special File"
  srcLight="/docs/light/route-special-file.png"
  srcDark="/docs/dark/route-special-file.png"
  width="1600"
  height="444"
/>

> **Good to know**: Route Handlers are only available inside the `app` directory. They are the equivalent of [API Routes](/docs/pages/building-your-application/routing/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](/docs/app/api-reference/file-conventions/route) inside the `app` directory:

```ts filename="app/api/route.ts" switcher
export const dynamic = 'force-dynamic' // defaults to force-static
export async function GET(request: Request) {}
```

```js filename="app/api/route.js" switcher
export const dynamic = 'force-dynamic' // defaults to force-static
export async function GET(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](https://developer.mozilla.org/docs/Web/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](https://developer.mozilla.org/docs/Web/API/Request) and [Response](https://developer.mozilla.org/docs/Web/API/Response). Next.js extends them with [`NextRequest`](/docs/app/api-reference/functions/next-request) and [`NextResponse`](/docs/app/api-reference/functions/next-response) to provide convenient helpers for advanced use cases.

## Behavior

### Caching

Route Handlers are cached by default when using the `GET` method with the `Response` object.

```ts filename="app/items/route.ts" switcher
export async function GET() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY,
    },
  })
  const data = await res.json()

  return Response.json({ data })
}
```

```js filename="app/items/route.js" switcher
export async function GET() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY,
    },
  })
  const data = await res.json()

  return Response.json({ data })
}
```

> **TypeScript Warning:** `Response.json()` is only valid from TypeScript 5.2. If you use a lower TypeScript version, you can use [`NextResponse.json()`](/docs/app/api-reference/functions/next-response#json) for typed responses instead.

### Opting out of caching

You can opt out of caching by:

- Using the `Request` object with the `GET` method.
- Using any of the other HTTP methods.
- Using [Dynamic Functions](#dynamic-functions) like `cookies` and `headers`.
- The [Segment Config Options](#segment-config-options) manually specifies dynamic mode.

For example:

```ts filename="app/products/api/route.ts" switcher
export async function GET(request: Request) {
  const { searchParams } = new URL(request.url)
  const id = searchParams.get('id')
  const res = await fetch(`https://data.mongodb-api.com/product/${id}`, {
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY!,
    },
  })
  const product = await res.json()

  return Response.json({ product })
}
```

```js filename="app/products/api/route.js" switcher
export async function GET(request) {
  const { searchParams } = new URL(request.url)
  const id = searchParams.get('id')
  const res = await fetch(`https://data.mongodb-api.com/product/${id}`, {
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY,
    },
  })
  const product = await res.json()

  return Response.json({ product })
}
```

Similarly, the `POST` method will cause the Route Handler to be evaluated dynamically.

```ts filename="app/items/route.ts" switcher
export async function POST() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY!,
    },
    body: JSON.stringify({ time: new Date().toISOString() }),
  })

  const data = await res.json()

  return Response.json(data)
}
```

```js filename="app/items/route.js" switcher
export async function POST() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY,
    },
    body: JSON.stringify({ time: new Date().toISOString() }),
  })

  const data = await res.json()

  return Response.json(data)
}
```

> **Good to know**: Like API Routes, Route Handlers can be used for cases like handling form submissions. A new abstraction for [handling forms and mutations](/docs/app/building-your-application/data-fetching/forms-and-mutations) that integrates deeply with React is being worked on.

### 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
`app/page.js`	`app/route.js`	<Cross size={18} /> Conflict
`app/page.js`	`app/api/route.js`	<Check size={18} /> Valid
`app/[user]/page.js`	`app/api/route.js`	<Check size={18} /> Valid

Each `route.js` or `page.js` file takes over all HTTP verbs for that route.

```jsx filename="app/page.js"
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}

// ❌ Conflict
// `app/route.js`
export async function POST(request) {}
```

## Examples

The following examples show how to combine Route Handlers with other Next.js APIs and features.

### Revalidating Cached Data

You can [revalidate cached data](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#revalidating-data) using the [`next.revalidate`](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#revalidating-data) option:

```ts filename="app/items/route.ts" switcher
export async function GET() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    next: { revalidate: 60 }, // Revalidate every 60 seconds
  })
  const data = await res.json()

  return Response.json(data)
}
```

```js filename="app/items/route.js" switcher
export async function GET() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    next: { revalidate: 60 }, // Revalidate every 60 seconds
  })
  const data = await res.json()

  return Response.json(data)
}
```

Alternatively, you can use the [`revalidate` segment config option](/docs/app/api-reference/file-conventions/route-segment-config#revalidate):

```ts
export const revalidate = 60
```

### Dynamic Functions

Route Handlers can be used with dynamic functions from Next.js, like [`cookies`](/docs/app/api-reference/functions/cookies) and [`headers`](/docs/app/api-reference/functions/headers).

#### Cookies

You can read cookies with [`cookies`](/docs/app/api-reference/functions/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`](https://developer.mozilla.org/docs/Web/HTTP/Headers/Set-Cookie) header.

```ts filename="app/api/route.ts" switcher
import { cookies } from 'next/headers'

export async function GET(request: Request) {
  const cookieStore = cookies()
  const token = cookieStore.get('token')

  return new Response('Hello, Next.js!', {
    status: 200,
    headers: { 'Set-Cookie': `token=${token.value}` },
  })
}
```

```js filename="app/api/route.js" switcher
import { cookies } from 'next/headers'

export async function GET(request) {
  const cookieStore = cookies()
  const token = cookieStore.get('token')

  return new Response('Hello, Next.js!', {
    status: 200,
    headers: { 'Set-Cookie': `token=${token}` },
  })
}
```

Alternatively, you can use abstractions on top of the underlying Web APIs to read cookies ([`NextRequest`](/docs/app/api-reference/functions/next-request)):

```ts filename="app/api/route.ts" switcher
import { type NextRequest } from 'next/server'

export async function GET(request: NextRequest) {
  const token = request.cookies.get('token')
}
```

```js filename="app/api/route.js" switcher
export async function GET(request) {
  const token = request.cookies.get('token')
}
```

#### Headers

You can read headers with [`headers`](/docs/app/api-reference/functions/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`.

```ts filename="app/api/route.ts" switcher
import { headers } from 'next/headers'

export async function GET(request: Request) {
  const headersList = headers()
  const referer = headersList.get('referer')

  return new Response('Hello, Next.js!', {
    status: 200,
    headers: { referer: referer },
  })
}
```

```js filename="app/api/route.js" switcher
import { headers } from 'next/headers'

export async function GET(request) {
  const headersList = headers()
  const referer = headersList.get('referer')

  return new Response('Hello, Next.js!', {
    status: 200,
    headers: { referer: referer },
  })
}
```

Alternatively, you can use abstractions on top of the underlying Web APIs to read headers ([`NextRequest`](/docs/app/api-reference/functions/next-request)):

```ts filename="app/api/route.ts" switcher
import { type NextRequest } from 'next/server'

export async function GET(request: NextRequest) {
  const requestHeaders = new Headers(request.headers)
```

}

```js filename="app/api/route.js" switcher
export async function GET(request) {
  const requestHeaders = new Headers(request.headers)
}
```

### Redirects

```ts filename="app/api/route.ts" switcher
import { redirect } from 'next/navigation'

export async function GET(request: Request) {
  redirect('https://nextjs.org/')
}
```

```js filename="app/api/route.js" switcher
import { redirect } from 'next/navigation'

export async function GET(request) {
  redirect('https://nextjs.org/')
}
```

### Dynamic Route Segments

> We recommend reading the [Defining Routes](/docs/app/building-your-application/routing/defining-routes) page before continuing.

Route Handlers can use [Dynamic Segments](/docs/app/building-your-application/routing/dynamic-routes) to create request handlers from dynamic data.

```ts filename="app/items/[slug]/route.ts" switcher
export async function GET(
  request: Request,
  { params }: { params: { slug: string } }
) {
  const slug = params.slug // 'a', 'b', or 'c'
}
```

```js filename="app/items/[slug]/route.js" switcher
export async function GET(request, { params }) {
```

  const slug = params.slug // 'a', 'b', or 'c'
}

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' }`

### URL Query Parameters

The request object passed to the Route Handler is a `NextRequest` instance, which has [some additional convenience methods](/docs/app/api-reference/functions/next-request#nexturl), including for more easily handling query parameters.

```ts filename="app/api/search/route.ts" switcher
import { type NextRequest } from 'next/server'

export function GET(request: NextRequest) {
  const searchParams = request.nextUrl.searchParams
  const query = searchParams.get('query')
  // query is "hello" for /api/search?query=hello
}
```

```js filename="app/api/search/route.js" switcher
export function GET(request) {
  const searchParams = request.nextUrl.searchParams
  const query = searchParams.get('query')
  // query is "hello" for /api/search?query=hello
}
```

### Streaming

Streaming is commonly used in combination with Large Language Models (LLMs), such as OpenAI, for AI-generated content. Learn more about the [AI SDK](https://sdk.vercel.ai/docs).

```ts filename="app/api/chat/route.ts" switcher
import OpenAI from 'openai'
import { OpenAIStream, StreamingTextResponse } from 'ai'

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
```

```ts
})

export const runtime = 'edge'

export async function POST(req: Request) {
  const { messages } = await req.json()
  const response = await openai.chat.completions.create({
    model: 'gpt-3.5-turbo',
    stream: true,
    messages,
  })

  const stream = OpenAIStream(response)

  return new StreamingTextResponse(stream)
}
```

```js filename="app/api/chat/route.js" switcher
import OpenAI from 'openai'
import { OpenAIStream, StreamingTextResponse } from 'ai'

const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY,
})

export const runtime = 'edge'

export async function POST(req) {
  const { messages } = await req.json()
  const response = await openai.chat.completions.create({
    model: 'gpt-3.5-turbo',
    stream: true,
    messages,
  })

  const stream = OpenAIStream(response)

  return new StreamingTextResponse(stream)
}
```

These abstractions use the Web APIs to create a stream. You can also use the underlying Web APIs directly.

```ts filename="app/api/route.ts" switcher
// https://developer.mozilla.org/docs/Web/API/
```

~~~~
ReadableStream#convert_async_iterator_to_stream
function iteratorToStream(iterator: any) {
  return new ReadableStream({
    async pull(controller) {
      const { value, done } = await iterator.next()

      if (done) {
        controller.close()
      } else {
        controller.enqueue(value)
      }
    },
  })
}

function sleep(time: number) {
  return new Promise((resolve) => {
    setTimeout(resolve, time)
  })
}

const encoder = new TextEncoder()

async function* makeIterator() {
  yield encoder.encode('<p>One</p>')
  await sleep(200)
  yield encoder.encode('<p>Two</p>')
  await sleep(200)
  yield encoder.encode('<p>Three</p>')
}

export async function GET() {
  const iterator = makeIterator()
  const stream = iteratorToStream(iterator)

  return new Response(stream)
}
```

```js filename="app/api/route.js" switcher
// https://developer.mozilla.org/docs/Web/API/
ReadableStream#convert_async_iterator_to_stream
function iteratorToStream(iterator) {
  return new ReadableStream({
    async pull(controller) {
      const { value, done } = await iterator.next()
~~~~

~~~markdown
      if (done) {
        controller.close()
      } else {
        controller.enqueue(value)
      }
    },
  })
}

function sleep(time) {
  return new Promise((resolve) => {
    setTimeout(resolve, time)
  })
}

const encoder = new TextEncoder()

async function* makeIterator() {
  yield encoder.encode('<p>One</p>')
  await sleep(200)
  yield encoder.encode('<p>Two</p>')
  await sleep(200)
  yield encoder.encode('<p>Three</p>')
}

export async function GET() {
  const iterator = makeIterator()
  const stream = iteratorToStream(iterator)

  return new Response(stream)
}
```

### Request Body

You can read the `Request` body using the standard Web API methods:

```ts filename="app/items/route.ts" switcher
export async function POST(request: Request) {
  const res = await request.json()
  return Response.json({ res })
}
```

```js filename="app/items/route.js" switcher
export async function POST(request) {
~~~

```
  const res = await request.json()
  return Response.json({ res })
}
```

### Request Body FormData

You can read the `FormData` using the `request.formData()` function:

```ts filename="app/items/route.ts" switcher
export async function POST(request: Request) {
  const formData = await request.formData()
  const name = formData.get('name')
  const email = formData.get('email')
  return Response.json({ name, email })
}
```

```js filename="app/items/route.js" switcher
export async function POST(request) {
  const formData = await request.formData()
  const name = formData.get('name')
  const email = formData.get('email')
  return Response.json({ name, email })
}
```

Since `formData` data are all strings, you may want to use [`zod-form-data`](https://www.npmjs.com/zod-form-data) to validate the request and retrieve data in the format you prefer (e.g. `number`).

### CORS

You can set CORS headers on a `Response` using the standard Web API methods:

```ts filename="app/api/route.ts" switcher
export const dynamic = 'force-dynamic' // defaults to force-static

export async function GET(request: Request) {
  return new Response('Hello, Next.js!', {
    status: 200,
    headers: {
      'Access-Control-Allow-Origin': '*',
      'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
      'Access-Control-Allow-Headers': 'Content-Type, Authorization',
    },
```

  })
}

```js filename="app/api/route.js" switcher
export const dynamic = 'force-dynamic' // defaults to force-static

export async function GET(request) {
  return new Response('Hello, Next.js!', {
    status: 200,
    headers: {
      'Access-Control-Allow-Origin': '*',
      'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
      'Access-Control-Allow-Headers': 'Content-Type, Authorization',
    },
  })
}
```

### 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](/docs/app/api-reference/file-conventions/route-segment-config) as Pages and Layouts, they support long-awaited features like general-purpose [statically regenerated](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#revalidating-data) Route Handlers.

You can use the `runtime` segment config option to specify the runtime:

```ts
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`](/docs/app/api-reference/file-conventions/metadata/sitemap#generate-a-sitemap), [`robots.txt`](/docs/app/api-reference/file-conventions/metadata/robots#generate-a-robots-file), [`app icons`](/docs/app/api-reference/file-conventions/metadata/app-icons#generate-icons-using-code-js-ts-tsx), and [open graph images](/docs/app/api-reference/file-conventions/metadata/opengraph-image) all have built-in support.

```ts filename="app/rss.xml/route.ts" switcher
export const dynamic = 'force-dynamic' // defaults to force-static
```

export async function GET() {
  return new Response(`<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
  <title>Next.js Documentation</title>
  <link>https://nextjs.org/docs</link>
  <description>The React Framework for the Web</description>
</channel>

</rss>`)
}

```js filename="app/rss.xml/route.js" switcher
export const dynamic = 'force-dynamic' // defaults to force-static

export async function GET() {
  return new Response(`<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
  <title>Next.js Documentation</title>
  <link>https://nextjs.org/docs</link>
  <description>The React Framework for the Web</description>
</channel>

</rss>`)
}
```

### Segment Config Options

Route Handlers use the same [route segment configuration](/docs/app/api-reference/file-conventions/route-segment-config) as pages and layouts.

```ts filename="app/items/route.ts" switcher
export const dynamic = 'auto'
export const dynamicParams = true
export const revalidate = false
export const fetchCache = 'auto'
export const runtime = 'nodejs'
export const preferredRegion = 'auto'
```

```js filename="app/items/route.js" switcher
```

```
export const dynamic = 'auto'
export const dynamicParams = true
export const revalidate = false
export const fetchCache = 'auto'
export const runtime = 'nodejs'
export const preferredRegion = 'auto'
```

See the [API reference](/docs/app/api-reference/file-conventions/route-segment-config) for more details.

---
title: Middleware
description: Learn how to use Middleware to run code before a request is completed.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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](#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 filename="middleware.ts" switcher
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

// This function can be marked `async` if using `await` inside
export function middleware(request: NextRequest) {
  return NextResponse.redirect(new URL('/home', request.url))
}

// See "Matching Paths" below to learn more
```

export const config = {
  matcher: '/about/:path*',
}

```js filename="middleware.js" switcher
import { NextResponse } from 'next/server'

// This function can be marked `async` if using `await` inside
export function middleware(request) {
  return NextResponse.redirect(new URL('/home', request.url))
}

// See "Matching Paths" below to learn more
export const config = {
  matcher: '/about/:path*',
}
```

## 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](#matcher)
2. [Conditional statements](#conditional-statements)

### Matcher

`matcher` allows you to filter Middleware to run on specific paths.

```js filename="middleware.js"
export const config = {
  matcher: '/about/:path*',
}
```

You can match a single path or multiple paths with an array syntax:

```js filename="middleware.js"
export const config = {
  matcher: ['/about/:path*', '/dashboard/:path*'],
}
```

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:

```js filename="middleware.js"
export const config = {
  matcher: [
    /*
     * Match all request paths except for the ones starting with:
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization files)
     * - favicon.ico (favicon file)
     */
    '/((?!api|_next/static|_next/image|favicon.ico).*)',
  ],
}
```

> **Good to know**: 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](https://github.com/pillarjs/path-to-regexp#path-to-regexp-1) documentation.

> **Good to know**: For backward compatibility, Next.js always considers `/

public` as `/public/index`. Therefore, a matcher of `/public/:path` will match.

### Conditional Statements

```ts filename="middleware.ts" switcher
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  if (request.nextUrl.pathname.startsWith('/about')) {
    return NextResponse.rewrite(new URL('/about-2', request.url))
  }

  if (request.nextUrl.pathname.startsWith('/dashboard')) {
    return NextResponse.rewrite(new URL('/dashboard/user', request.url))
  }
}
```

```js filename="middleware.js" switcher
import { NextResponse } from 'next/server'

export function middleware(request) {
  if (request.nextUrl.pathname.startsWith('/about')) {
    return NextResponse.rewrite(new URL('/about-2', request.url))
  }

  if (request.nextUrl.pathname.startsWith('/dashboard')) {
    return NextResponse.rewrite(new URL('/dashboard/user', request.url))
  }
}
```

## 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

<AppOnly>

To produce a response from Middleware, you can:

1. `rewrite` to a route ([Page](/docs/app/building-your-application/routing/pages-and-layouts) or [Route Handler](/docs/app/building-your-application/routing/route-handlers)) that produces a response
2. return a `NextResponse` directly. See [Producing a Response](#producing-a-response)

</AppOnly>

<PagesOnly>

To produce a response from Middleware, you can:

1. `rewrite` to a route ([Page](/docs/pages/building-your-application/routing/pages-and-layouts) or [Edge API Route](/docs/pages/building-your-application/routing/api-routes)) that produces a response
2. return a `NextResponse` directly. See [Producing a Response](#producing-a-response)

</PagesOnly>

## 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 filename="middleware.ts" switcher
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  // Assume a "Cookie:nextjs=fast" header to be present on the incoming request
  // Getting cookies from the request using the `RequestCookies` API
  let cookie = request.cookies.get('nextjs')
  console.log(cookie) // => { name: 'nextjs', value: 'fast', Path: '/' }
  const allCookies = request.cookies.getAll()
  console.log(allCookies) // => [{ name: 'nextjs', value: 'fast' }]
```

~~~~
  request.cookies.has('nextjs') // => true
  request.cookies.delete('nextjs')
  request.cookies.has('nextjs') // => false

  // Setting cookies on the response using the `ResponseCookies` API
  const response = NextResponse.next()
  response.cookies.set('vercel', 'fast')
  response.cookies.set({
    name: 'vercel',
    value: 'fast',
    path: '/',
  })
  cookie = response.cookies.get('vercel')
  console.log(cookie) // => { name: 'vercel', value: 'fast', Path: '/' }
  // The outgoing response will have a `Set-Cookie:vercel=fast;path=/test`
header.

  return response
}
```

```js filename="middleware.js" switcher
import { NextResponse } from 'next/server'

export function middleware(request) {
  // Assume a "Cookie:nextjs=fast" header to be present on the incoming
request
  // Getting cookies from the request using the `RequestCookies` API
  let cookie = request.cookies.get('nextjs')
  console.log(cookie) // => { name: 'nextjs', value: 'fast', Path: '/' }
  const allCookies = request.cookies.getAll()
  console.log(allCookies) // => [{ name: 'nextjs', value: 'fast' }]

  request.cookies.has('nextjs') // => true
  request.cookies.delete('nextjs')
  request.cookies.has('nextjs') // => false

  // Setting cookies on the response using the `ResponseCookies` API
  const response = NextResponse.next()
  response.cookies.set('vercel', 'fast')
  response.cookies.set({
    name: 'vercel',
    value: 'fast',
    path: '/',
  })
  cookie = response.cookies.get('vercel')
~~~~

  console.log(cookie) // => { name: 'vercel', value: 'fast', Path: '/' }
  // The outgoing response will have a `Set-Cookie:vercel=fast;path=/test`
header.

  return response
}

## 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 filename="middleware.ts" switcher
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  // Clone the request headers and set a new header `x-hello-from-
middleware1`
  const requestHeaders = new Headers(request.headers)
  requestHeaders.set('x-hello-from-middleware1', 'hello')

  // You can also set request headers in NextResponse.rewrite
  const response = NextResponse.next({
    request: {
      // New request headers
      headers: requestHeaders,
    },
  })

  // Set a new response header `x-hello-from-middleware2`
  response.headers.set('x-hello-from-middleware2', 'hello')
  return response
}
```

```js filename="middleware.js" switcher
import { NextResponse } from 'next/server'

export function middleware(request) {
  // Clone the request headers and set a new header `x-hello-from-
middleware1`
  const requestHeaders = new Headers(request.headers)
  requestHeaders.set('x-hello-from-middleware1', 'hello')

  // You can also set request headers in NextResponse.rewrite
```

  const response = NextResponse.next({
    request: {
      // New request headers
      headers: requestHeaders,
    },
  })

  // Set a new response header `x-hello-from-middleware2`
  response.headers.set('x-hello-from-middleware2', 'hello')
  return response
}

> **Good to know**: Avoid setting large headers as it might cause [431 Request Header Fields Too Large](https://developer.mozilla.org/docs/Web/HTTP/Status/431) 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](https://nextjs.org/blog/next-13-1#nextjs-advanced-middleware))

```ts filename="middleware.ts" switcher
import { NextRequest } from 'next/server'
import { isAuthenticated } from '@lib/auth'

// Limit the middleware to paths starting with `/api/`
export const config = {
  matcher: '/api/:function*',
}

export function middleware(request: NextRequest) {
  // Call our authentication function to check the request
  if (!isAuthenticated(request)) {
    // Respond with JSON indicating an error message
    return Response.json(
      { success: false, message: 'authentication failed' },
      { status: 401 }
    )
  }
}
```

```js filename="middleware.js" switcher
import { isAuthenticated } from '@lib/auth'
```

```
// Limit the middleware to paths starting with `/api/`
export const config = {
  matcher: '/api/:function*',
}

export function middleware(request) {
  // Call our authentication function to check the request
  if (!isAuthenticated(request)) {
    // Respond with JSON indicating an error message
    return Response.json(
      { success: false, message: 'authentication failed' },
      { status: 401 }
    )
  }
}
```

## 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 filename="next.config.js"
module.exports = {
  skipTrailingSlashRedirect: true,
}
```

```js filename="middleware.js"
const legacyPrefixes = ['/docs', '/blog']

export default async function middleware(req) {
  const { pathname } = req.nextUrl

  if (legacyPrefixes.some((prefix) => pathname.startsWith(prefix))) {
    return NextResponse.next()
  }

  // apply trailing slash handling
  if (
```

    !pathname.endsWith('/') &&
    !pathname.match(/((?!|.well-known(?:|/.*)?)(?:[^/]+|/)*[^/]+|.|w+)/)
) {
    req.nextUrl.pathname += '/'
    return NextResponse.redirect(req.nextUrl)
  }
}

`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 filename="next.config.js"
module.exports = {
  skipMiddlewareUrlNormalize: true,
}
```

```js filename="middleware.js"
export default async function middleware(req) {
  const { pathname } = req.nextUrl

  // GET /_next/data/build-id/hello.json

  console.log(pathname)
  // with the flag this now /_next/data/build-id/hello.json
  // without the flag this would be normalized to /hello
}
```

## Runtime

Middleware currently only supports the [Edge runtime](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes). The Node.js runtime can not be used.

## 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](/docs/messages/middleware-upgrade-guide) |
| `v12.0.9` | Enforce absolute URLs in Edge Runtime ([PR](https://github.com/vercel/next.js/pull/33410)) |
| `v12.0.0` | Middleware (Beta) added
|

---
title: Project Organization and File Colocation
nav_title: Project Organization
description: Learn how to organize your Next.js project and colocate files.
related:
  links:
    - app/building-your-application/routing/defining-routes
    - app/building-your-application/routing/route-groups
    - app/building-your-application/configuring/src-directory
    - app/building-your-application/configuring/absolute-imports-and-module-aliases
---

Apart from [routing folder and file conventions](/docs/getting-started/project-structure#app-routing-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](#safe-colocation-by-default)
- [Project organization features](#project-organization-features)
- [Project organization strategies](#project-organization-strategies)

## Safe colocation by default

In the `app` directory, [nested folder hierarchy](/docs/app/building-your-application/routing#route-segments) 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 publicly accessible** until a `page.js` or `route.js` file is added to a route segment.

<Image
  alt="A diagram showing how a route is not publicly accessible until a page.js or route.js file is added to a route segment."

  srcLight="/docs/light/project-organization-not-routable.png"
  srcDark="/docs/dark/project-organization-not-routable.png"
  width="1600"
  height="444"
/>

And, even when a route is made publicly accessible, only the **content returned** by `page.js` or `route.js` is sent to the client.

<Image
  alt="A diagram showing how page.js and route.js files make routes publicly accessible."
  srcLight="/docs/light/project-organization-routable.png"
  srcDark="/docs/dark/project-organization-routable.png"
  width="1600"
  height="687"
/>

This means that **project files** can be **safely colocated** inside route segments in the `app` directory without accidentally being routable.

<Image
  alt="A diagram showing colocated project files are not routable even when a segment contains a page.js or route.js file."
  srcLight="/docs/light/project-organization-colocation.png"
  srcDark="/docs/dark/project-organization-colocation.png"
  width="1600"
  height="1011"
/>

> **Good to know**:
>
> - This is different from the `pages` directory, where any file in `pages` is considered a route.
> - While you **can** colocate your project files in `app` you don't **have** to. If you prefer, you can [keep them outside the `app` directory](#store-project-files-outside-of-app).

## 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.

<Image
  alt="An example folder structure using private folders"
  srcLight="/docs/light/project-organization-private-folders.png"
  srcDark="/docs/dark/project-organization-private-folders.png"
  width="1600"
  height="849"
/>

Since files in the `app` directory can be [safely colocated by default](#safe-colocation-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](/docs/getting-started/project-structure#routing-files) 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.

<Image
  alt="An example folder structure using route groups"
  srcLight="/docs/light/project-organization-route-groups.png"
  srcDark="/docs/dark/project-organization-route-groups.png"
  width="1600"

  height="849"
/>

Route groups are useful for:

- [Organizing routes into groups](/docs/app/building-your-application/routing/route-groups#organize-routes-without-affecting-the-url-path) 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](/docs/app/building-your-application/routing/route-groups#creating-multiple-root-layouts)
  - [Adding a layout to a subset of routes in a common segment](/docs/app/building-your-application/routing/route-groups#opting-specific-segments-into-a-layout)

### `src` Directory

Next.js supports storing application code (including `app`) inside an optional [`src` directory](/docs/app/building-your-application/configuring/src-directory). This separates application code from project configuration files which mostly live in the root of a project.

<Image
  alt="An example folder structure with the `src` directory"
  srcLight="/docs/light/project-organization-src-directory.png"
  srcDark="/docs/dark/project-organization-src-directory.png"
  width="1600"
  height="687"
/>

### Module Path Aliases

Next.js supports [Module Path Aliases](/docs/app/building-your-application/configuring/absolute-imports-and-module-aliases) which make it easier to read and maintain imports across deeply nested project files.

```jsx filename="app/dashboard/settings/analytics/page.js"
// before
import { Button } from '../../../components/button'

// after
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 a Next.js 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.

> **Good to know**: In our examples below, we're using `components` and `lib` folders as generalized placeholders, their naming has no special framework significance and your projects might use other folders like `ui`, `utils`, `hooks`, `styles`, etc.

### 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.

<Image
  alt="An example folder structure with project files outside of app"
  srcLight="/docs/light/project-organization-project-root.png"
  srcDark="/docs/dark/project-organization-project-root.png"
  width="1600"
  height="849"
/>

### 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**.

<Image
  alt="An example folder structure with project files inside app"
  srcLight="/docs/light/project-organization-app-root.png"
  srcDark="/docs/dark/project-organization-app-root.png"
  width="1600"
  height="849"
/>

### 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.

<Image
  alt="An example folder structure with project files split by feature or route"

  srcLight="/docs/light/project-organization-app-root-split.png"
  srcDark="/docs/dark/project-organization-app-root-split.png"
  width="1600"
  height="1011"
/>

---
title: Internationalization
description: Add support for multiple languages with internationalized routing and localized content.

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 filename="middleware.js"
import { match } from '@formatjs/intl-localematcher'
import Negotiator from 'negotiator'

let headers = { 'accept-language': 'en-US,en;q=0.5' }
let languages = new Negotiator({ headers }).languages()
let locales = ['en-US', 'nl-NL', 'nl']
let defaultLocale = 'en-US'

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](/docs/app/building-your-application/routing/middleware).

```js filename="middleware.js"

let locales = ['en-US', 'nl-NL', 'nl']

// Get the preferred locale, similar to the above or using a library
function getLocale(request) { ... }

export function middleware(request) {
  // Check if there is any supported locale in the pathname
  const { pathname } = request.nextUrl
  const pathnameHasLocale = locales.some(
    (locale) => pathname.startsWith(`/${locale}/`) || pathname === `/${locale}`
  )

  if (pathnameHasLocale) return

  // Redirect if there is no locale
  const locale = getLocale(request)
  request.nextUrl.pathname = `/${locale}${pathname}`
  // e.g. incoming request is /products
  // The new URL is now /en-US/products
  return Response.redirect(request.nextUrl)
}

export const config = {
  matcher: [
    // Skip all internal paths (_next)
    '/((?!_next).*)',
    // Optional: only run on root (/) URL
    // '/'
  ],
}
```

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:

```jsx filename="app/[lang]/page.js"
// You now have access to the current locale
// e.g. /en-US/products -> `lang` is "en-US"
```

export default async function Page({ params: { lang } }) {
  return ...
}

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:

```json filename="dictionaries/en.json"
{
  "products": {
    "cart": "Add to Cart"
  }
}
```

```json filename="dictionaries/nl.json"
{
  "products": {
    "cart": "Toevoegen aan Winkelwagen"
  }
}
```

We can then create a `getDictionary` function to load the translations for the requested locale:

```jsx filename="app/[lang]/dictionaries.js"
import 'server-only'

const dictionaries = {
  en: () => import('./dictionaries/en.json').then((module) => module.default),
  nl: () => import('./dictionaries/nl.json').then((module) => module.default),
}

export const getDictionary = async (locale) => dictionaries[locale]()
```

Given the currently selected language, we can fetch the dictionary inside of a layout or page.

```jsx filename="app/[lang]/page.js"
import { getDictionary } from './dictionaries'

export default async function Page({ params: { lang } }) {
  const dict = await getDictionary(lang) // en
  return <button>{dict.products.cart}</button> // Add to Cart
}
```

Because all layouts and pages in the `app/` directory default to [Server Components](/docs/app/building-your-application/rendering/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:

```jsx filename="app/[lang]/layout.js"
export async function generateStaticParams() {
  return [{ lang: 'en-US' }, { lang: 'de' }]
}

export default function Root({ children, params }) {
  return (
    <html lang={params.lang}>
      <body>{children}</body>
    </html>
  )
}
```

## Resources

- [Minimal i18n routing and translations](https://github.com/vercel/next.js/tree/canary/examples/app-dir-i18n-routing)
- [`next-intl`](https://next-intl-docs.vercel.app/docs/next-13)
- [`next-international`](https://github.com/QuiiBz/next-international)
- [`next-i18n-router`](https://github.com/i18nexus/next-i18n-router)
---

title: Data Fetching, Caching, and Revalidating
nav_title: Fetching, Caching, and Revalidating
description: Learn how to fetch, cache, and revalidate data in your Next.js application.

Data fetching is a core part of any application. This page goes through how you can fetch, cache, and revalidate data in React and Next.js.

There are four ways you can fetch data:

1. [On the server, with `fetch`](#fetching-data-on-the-server-with-fetch)
2. [On the server, with third-party libraries](#fetching-data-on-the-server-with-third-party-libraries)
3. [On the client, via a Route Handler](#fetching-data-on-the-client-with-route-handlers)
4. [On the client, with third-party libraries](#fetching-data-on-the-client-with-route-handlers).

## Fetching Data on the Server with `fetch`

Next.js extends the native [`fetch` Web API](https://developer.mozilla.org/docs/Web/API/Fetch_API) to allow you to configure the [caching](#caching-data) and [revalidating](#revalidating-data) behavior for each fetch request on the server. React extends `fetch` to automatically [memoize](/docs/app/building-your-application/data-fetching/patterns#fetching-data-where-its-needed) fetch requests while rendering a React component tree.

You can use `fetch` with `async`/`await` in Server Components, in [Route Handlers](/docs/app/building-your-application/routing/route-handlers), and in [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations).

For example:

```tsx filename="app/page.tsx" switcher
async function getData() {
  const res = await fetch('https://api.example.com/...')
  // The return value is *not* serialized
  // You can return Date, Map, Set, etc.

  if (!res.ok) {
    // This will activate the closest `error.js` Error Boundary
    throw new Error('Failed to fetch data')
  }

  return res.json()
```

}

export default async function Page() {
  const data = await getData()

  return <main></main>
}

```jsx filename="app/page.js" switcher
async function getData() {
  const res = await fetch('https://api.example.com/...')
  // The return value is *not* serialized
  // You can return Date, Map, Set, etc.

  if (!res.ok) {
    // This will activate the closest `error.js` Error Boundary
    throw new Error('Failed to fetch data')
  }

  return res.json()
}

export default async function Page() {
  const data = await getData()

  return <main></main>
}
```

> **Good to know**:
>
> - Next.js provides helpful functions you may need when fetching data in Server Components such as [`cookies`](/docs/app/api-reference/functions/cookies) and [`headers`](/docs/app/api-reference/functions/headers). These will cause the route to be dynamically rendered as they rely on request time information.
> - In Route handlers, `fetch` requests are not memoized as Route Handlers are not part of the React component tree.
> - To use `async`/`await` in a Server Component with TypeScript, you'll need to use TypeScript `5.1.3` or higher and `@types/react` `18.2.8` or higher.

### Caching Data

Caching stores data so it doesn't need to be re-fetched from your data source on every request.

By default, Next.js automatically caches the returned values of `fetch` in the [Data Cache](/docs/app/building-your-application/caching#data-cache) on the server. This means that the data can be fetched at build time or request time, cached, and reused on each data request.

```js
// 'force-cache' is the default, and can be omitted
fetch('https://...', { cache: 'force-cache' })
```

`fetch` requests that use the `POST` method are also automatically cached. Unless it's inside a [Route Handler](/docs/app/building-your-application/routing/route-handlers) that uses the `POST` method, then it will not be cached.

> **What is the Data Cache?**
>
> The Data Cache is a persistent [HTTP cache](https://developer.mozilla.org/docs/Web/HTTP/Caching). Depending on your platform, the cache can scale automatically and be [shared across multiple regions](https://vercel.com/docs/infrastructure/data-cache).
>
> Learn more about the [Data Cache](/docs/app/building-your-application/caching#data-cache).

### Revalidating Data

Revalidation is the process of purging the Data Cache and re-fetching the latest data. This is useful when your data changes and you want to ensure you show the latest information.

Cached data can be revalidated in two ways:

- **Time-based revalidation**: Automatically revalidate data after a certain amount of time has passed. This is useful for data that changes infrequently and freshness is not as critical.
- **On-demand revalidation**: Manually revalidate data based on an event (e.g. form submission). On-demand revalidation can use a tag-based or path-based approach to revalidate groups of data at once. This is useful when you want to ensure the latest data is shown as soon as possible (e.g. when content from your headless CMS is updated).

#### Time-based Revalidation

To revalidate data at a timed interval, you can use the `next.revalidate` option of `fetch` to set the cache lifetime of a resource (in seconds).

```js
fetch('https://...', { next: { revalidate: 3600 } })
```

Alternatively, to revalidate all `fetch` requests in a route segment, you can use the [Segment Config Options](/docs/app/api-reference/file-conventions/route-segment-config).

```jsx filename="layout.js | page.js"
export const revalidate = 3600 // revalidate at most every hour
```

If you have multiple fetch requests in a statically rendered route, and each has a different revalidation frequency. The lowest time will be used for all requests. For dynamically rendered routes, each `fetch` request will be revalidated independently.

Learn more about [time-based revalidation](/docs/app/building-your-application/caching#time-based-revalidation).

#### On-demand Revalidation

Data can be revalidated on-demand by path ([`revalidatePath`](/docs/app/api-reference/functions/revalidatePath)) or by cache tag ([`revalidateTag`](/docs/app/api-reference/functions/revalidateTag)) inside a [Server Action](/docs/app/building-your-application/data-fetching/forms-and-mutations) or [Route Handler](/docs/app/building-your-application/routing/route-handlers).

Next.js has a cache tagging system for invalidating `fetch` requests across routes.

1. When using `fetch`, you have the option to tag cache entries with one or more tags.
2. Then, you can call `revalidateTag` to revalidate all entries associated with that tag.

For example, the following `fetch` request adds the cache tag `collection`:

```tsx filename="app/page.tsx" switcher
export default async function Page() {
  const res = await fetch('https://...', { next: { tags: ['collection'] } })
  const data = await res.json()
  // ...
}
```

```jsx filename="app/page.js" switcher
export default async function Page() {
  const res = await fetch('https://...', { next: { tags: ['collection'] } })
  const data = await res.json()
  // ...
}
```

You can then revalidate this `fetch` call tagged with `collection` by calling `revalidateTag` in a Server Action:

```ts filename="app/actions.ts" switcher
'use server'

import { revalidateTag } from 'next/cache'

export default async function action() {
  revalidateTag('collection')
}
```

```js filename="app/actions.js" switcher
'use server'

import { revalidateTag } from 'next/cache'

export default async function action() {
  revalidateTag('collection')
}
```

Learn more about [on-demand revalidation](/docs/app/building-your-application/caching#on-demand-revalidation).

#### 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.

### Opting out of Data Caching

`fetch` requests are **not** cached if:

- The `cache: 'no-store'` is added to `fetch` requests.
- The `revalidate: 0` option is added to individual `fetch` requests.

- The `fetch` request is inside a Router Handler that uses the `POST` method.
- The `fetch` request comes after the usage of `headers` or `cookies`.
- The `const dynamic = 'force-dynamic'` route segment option is used.
- The `fetchCache` route segment option is configured to skip cache by default.
- The `fetch` request uses `Authorization` or `Cookie` headers and there's an uncached request above it in the component tree.

#### Individual `fetch` Requests

To opt out of caching for individual `fetch` requests, you can set the `cache` option in `fetch` to `'no-store'`. This will fetch data dynamically, on every request.

```js filename="layout.js | page.js"
fetch('https://...', { cache: 'no-store' })
```

View all the available `cache` options in the [`fetch` API reference](/docs/app/api-reference/functions/fetch).

#### Multiple `fetch` Requests

If you have multiple `fetch` requests in a route segment (e.g. a Layout or Page), you can configure the caching behavior of all data requests in the segment using the [Segment Config Options](/docs/app/api-reference/file-conventions/route-segment-config).

However, we recommend configuring the caching behavior of each `fetch` request individually. This gives you more granular control over the caching behavior.

## Fetching data on the Server with third-party libraries

In cases where you're using a third-party library that doesn't support or expose `fetch` (for example, a database, CMS, or ORM client), you can configure the caching and revalidating behavior of those requests using the [Route Segment Config Option](/docs/app/api-reference/file-conventions/route-segment-config) and React's `cache` function.

Whether the data is cached or not will depend on whether the route segment is [statically or dynamically rendered](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies). If the segment is static (default), the output of the request will be cached and revalidated as part of the route 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.

You can also use the experimental [`unstable_cache` API](/docs/app/api-reference/functions/unstable_cache).

### Example

In the example below:

- The React `cache` function is used to [memoize](/docs/app/building-your-application/caching#request-memoization) data requests.
- The `revalidate` option is set to `3600` in the `layout.ts` and `page.ts` segments, meaning the data will be cached and revalidated at most every hour.

```ts filename="app/utils.ts" switcher
import { cache } from 'react'

export const getItem = cache(async (id: string) => {
  const item = await db.item.findUnique({ id })
  return item
})
```

```js filename="app/utils.js" switcher
import { cache } from 'react'

export const getItem = cache(async (id) => {
  const item = await db.item.findUnique({ id })
  return item
})
```

Although the `getItem` function is called twice, only one query will be made to the database.

```tsx filename="app/item/[id]/layout.tsx" switcher
import { getItem } from '@/utils/get-item'

export const revalidate = 3600 // revalidate the data at most every hour

export default async function Layout({
  params: { id },
}: {
  params: { id: string }
}) {
  const item = await getItem(id)
  // ...
}
```

```jsx filename="app/item/[id]/layout.js" switcher
import { getItem } from '@/utils/get-item'

export const revalidate = 3600 // revalidate the data at most every hour

export default async function Layout({ params: { id } }) {
  const item = await getItem(id)
  // ...
}
```

```tsx filename="app/item/[id]/page.tsx" switcher
import { getItem } from '@/utils/get-item'

export const revalidate = 3600 // revalidate the data at most every hour

export default async function Page({
  params: { id },
}: {
  params: { id: string }
}) {
  const item = await getItem(id)
  // ...
}
```

```jsx filename="app/item/[id]/page.js" switcher
import { getItem } from '@/utils/get-item'

export const revalidate = 3600 // revalidate the data at most every hour

export default async function Page({ params: { id } }) {
  const item = await getItem(id)
  // ...
}
```

## Fetching Data on the Client with Route Handlers

If you need to fetch data in a client component, you can call a [Route Handler](/docs/app/building-your-application/routing/route-handlers) from the client. Route Handlers execute on the server and return the data to the client. This is useful when you don't want to expose sensitive information to the client, such as API tokens.

See the [Route Handler](/docs/app/building-your-application/routing/route-handlers) documentation for examples.

> **Server Components and Route Handlers**
>
> Since Server Components render on the server, you don't need to call a Route Handler from a Server Component to fetch data. Instead, you can fetch the data directly inside the Server Component.

## Fetching Data on the Client with third-party libraries

You can also fetch data on the client using a third-party library such as [SWR](https://swr.vercel.app/) or [React Query](https://tanstack.com/query/latest). These libraries provide their own APIs for memoizing requests, caching, revalidating, and mutating data.

> **Future APIs**:
>
> `use` is a React function that **accepts and handles a promise** returned by a function. Wrapping `fetch` in `use` is currently **not** recommended in Client Components and may trigger multiple re-renders. Learn more about `use` in the [React docs](https://react.dev/reference/react/use).

---
title: Data Fetching Patterns
description: Learn about common data fetching patterns in React and Next.js.

There are a few recommended patterns and best practices for fetching data in React and Next.js. This page will go over some of the most common patterns and how to use them.

### Fetching Data on the Server

Whenever possible, we recommend fetching 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](https://vercel.com/blog/how-react-18-improves-application-performance) on the client.
- Perform multiple data fetches with single round-trip instead of multiple individual requests on the client.

- Reduce client-server [waterfalls](#parallel-and-sequential-data-fetching).
- Depending on your region, data fetching can also happen closer to your data source, reducing latency and improving performance.

You can fetch data on the server using Server Components, [Route Handlers](/docs/app/building-your-application/routing/route-handlers), and [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations).

### Fetching Data Where It's Needed

If you need to use the same data (e.g. current user) in multiple components in a tree, you do not have to fetch data globally, nor forward props between components. Instead, you can use `fetch` or React `cache` in the component that needs the data without worrying about the performance implications of making multiple requests for the same data.

This is possible because `fetch` requests are automatically memoized. Learn more about [request memoization](/docs/app/building-your-application/caching#request-memoization)

> **Good to know**: This also applies to layouts, since it's not possible to pass data between a parent layout and its children.

### Streaming

Streaming and [Suspense](https://react.dev/reference/react/Suspense) are 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](/docs/app/building-your-application/routing/pages-and-layouts), you're able to instantly render parts of the page that do not specifically require data, and show a [loading state](/docs/app/building-your-application/routing/loading-ui-and-streaming) 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.

<Image
  alt="Server Rendering with Streaming"
  srcLight="/docs/light/server-rendering-with-streaming.png"
  srcDark="/docs/dark/server-rendering-with-streaming.png"
  width="1600"
  height="785"
/>

To learn more about Streaming and Suspense, see the [Loading UI](/docs/app/building-your-application/routing/loading-ui-and-streaming) and [Streaming and Suspense](/docs/app/building-your-application/routing/loading-ui-and-streaming#streaming-with-suspense) pages.

### Parallel and Sequential Data Fetching

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

<Image
  alt="Sequential and Parallel Data Fetching"
  srcLight="/docs/light/sequential-parallel-data-fetching.png"
  srcDark="/docs/dark/sequential-parallel-data-fetching.png"
  width="1600"
  height="525"
/>

- With **sequential data fetching**, requests in a route are dependent on each other and therefore 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.
- 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.

#### Sequential Data Fetching

If you have nested components, and each component fetches its own data, then data fetching will happen sequentially if those data requests are different (this doesn't apply to requests for the same data as they are automatically [memoized](/docs/app/building-your-application/caching#request-memoization)).

For example, the `Playlists` component will only start fetching data once the `Artist` component has finished fetching data because `Playlists` depends on the `artistID` prop:

```tsx filename="app/artist/[username]/page.tsx" switcher
// ...

async function Playlists({ artistID }: { artistID: string }) {
  // Wait for the playlists
  const playlists = await getArtistPlaylists(artistID)

  return (
    <ul>
      {playlists.map((playlist) => (
```

~~~~
        <li key={playlist.id}>{playlist.name}</li>
      ))}
    </ul>
  )
}

export default async function Page({
  params: { username },
}: {
  params: { username: string }
}) {
  // Wait for the artist
  const artist = await getArtist(username)

  return (
    <>
      <h1>{artist.name}</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <Playlists artistID={artist.id} />
      </Suspense>
    </>
  )
}
```

```jsx filename="app/artist/[username]/page.js" switcher
// ...

async function Playlists({ artistID }) {
  // Wait for the playlists
  const playlists = await getArtistPlaylists(artistID)

  return (
    <ul>
      {playlists.map((playlist) => (
        <li key={playlist.id}>{playlist.name}</li>
      ))}
    </ul>
  )
}

export default async function Page({ params: { username } }) {
  // Wait for the artist
  const artist = await getArtist(username)

  return (
    <>
~~~~

      <h1>{artist.name}</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <Playlists artistID={artist.id} />
      </Suspense>
    </>
)
}
~~~

In cases like this, you can use [`loading.js`](/docs/app/building-your-application/routing/loading-ui-and-streaming) (for route segments) or [React `<Suspense>`](/docs/app/building-your-application/routing/loading-ui-and-streaming#streaming-with-suspense) (for nested components) to show an instant loading state while React streams in the result.

This will prevent the whole route from being blocked by data fetching, and the user will be able to interact with the parts of the page that are not blocked.

> **Blocking Data Requests:**
>
> An alternative approach to prevent waterfalls is to fetch data globally, at the root of your application, but this will block rendering for all route segments beneath it until the data has finished loading. This can be described as "all or nothing" data fetching. Either you have the entire data for your page or application, or none.
>
> Any fetch requests with `await` will block rendering and data fetching for the entire tree beneath it, unless they are wrapped in a `<Suspense>` boundary or `loading.js` is used. Another alternative is to use [parallel data fetching](#parallel-data-fetching) or the [preload pattern](#preloading-data).

Parallel Data Fetching

To fetch data in parallel, you can eagerly initiate requests by defining them outside the components that use the data, then calling them from inside the component. This saves time by initiating both requests in parallel, however, the user won't see the rendered result until both promises are resolved.

In the example below, the `getArtist` and `getArtistAlbums` functions are defined outside the `Page` component, then called inside the component, and we wait for both promises to resolve:

```tsx filename="app/artist/[username]/page.tsx" switcher
import Albums from './albums'

async function getArtist(username: string) {
  const res = await fetch(`https://api.example.com/artist/${username}`)
~~~

~~~
  return res.json()
}

async function getArtistAlbums(username: string) {
  const res = await fetch(`https://api.example.com/artist/${username}/albums`)
  return res.json()
}

export default async function Page({
  params: { username },
}: {
  params: { username: string }
}) {
  // Initiate both requests in parallel
  const artistData = getArtist(username)
  const albumsData = getArtistAlbums(username)

  // Wait for the promises to resolve
  const [artist, albums] = await Promise.all([artistData, albumsData])

  return (
    <>
      <h1>{artist.name}</h1>
      <Albums list={albums}></Albums>
    </>
  )
}
```

```jsx filename="app/artist/[username]/page.js" switcher
import Albums from './albums'

async function getArtist(username) {
  const res = await fetch(`https://api.example.com/artist/${username}`)
  return res.json()
}

async function getArtistAlbums(username) {
  const res = await fetch(`https://api.example.com/artist/${username}/albums`)
  return res.json()
}

export default async function Page({ params: { username } }) {
  // Initiate both requests in parallel
  const artistData = getArtist(username)
  const albumsData = getArtistAlbums(username)
~~~

```
  // Wait for the promises to resolve
  const [artist, albums] = await Promise.all([artistData, albumsData])

  return (
    <>
      <h1>{artist.name}</h1>
      <Albums list={albums}></Albums>
    </>
  )
}
```

To improve the user experience, you can add a [Suspense Boundary](/docs/app/building-your-application/routing/loading-ui-and-streaming) to break up the rendering work and show part of the result as soon as possible.

### Preloading Data

Another way to prevent waterfalls is to use the preload pattern. You can optionally create a `preload` function to further optimize parallel data fetching. With this approach, you don't have to pass promises down as props. The `preload` function can also have any name as it's a pattern, not an API.

```tsx filename="components/Item.tsx" switcher
import { getItem } from '@/utils/get-item'

export const preload = (id: string) => {
  // void evaluates the given expression and returns undefined
  // https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/void
  void getItem(id)
}
export default async function Item({ id }: { id: string }) {
  const result = await getItem(id)
  // ...
}
```

```jsx filename="components/Item.js" switcher
import { getItem } from '@/utils/get-item'

export const preload = (id) => {
  // void evaluates the given expression and returns undefined
  // https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/void
  void getItem(id)
}
export default async function Item({ id }) {
```

~~~
  const result = await getItem(id)
  // ...
}
```

```tsx filename="app/item/[id]/page.tsx" switcher
import Item, { preload, checkIsAvailable } from '@/components/Item'

export default async function Page({
  params: { id },
}: {
  params: { id: string }
}) {
  // starting loading item data
  preload(id)
  // perform another asynchronous task
  const isAvailable = await checkIsAvailable()

  return isAvailable ? <Item id={id} /> : null
}
```

```jsx filename="app/item/[id]/page.js" switcher
import Item, { preload, checkIsAvailable } from '@/components/Item'

export default async function Page({ params: { id } }) {
  // starting loading item data
  preload(id)
  // perform another asynchronous task
  const isAvailable = await checkIsAvailable()

  return isAvailable ? <Item id={id} /> : null
}
```

### Using React `cache`, `server-only`, and the Preload Pattern

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.

```ts filename="utils/get-item.ts" switcher
import { cache } from 'react'
import 'server-only'

export const preload = (id: string) => {
  void getItem(id)
~~~

}

export const getItem = cache(async (id: string) => {
  // ...
})

```js filename="utils/get-item.js" switcher
import { cache } from 'react'
import 'server-only'

export const preload = (id) => {
  void getItem(id)
}

export const getItem = cache(async (id) => {
  // ...
})
```

With this approach, you can eagerly fetch data, cache responses, and guarantee that this data fetching [only happens on the server](/docs/app/building-your-application/rendering/composition-patterns#keeping-server-only-code-out-of-the-client-environment).

The `utils/get-item` exports can be used by Layouts, Pages, or other components to give them control over when an item's data is fetched.

> **Good to know:**
>
> - We recommend using the [`server-only` package](/docs/app/building-your-application/rendering/composition-patterns#keeping-server-only-code-out-of-the-client-environment) to make sure server data fetching functions are never used on the client.

---
title: Forms and Mutations
nav_title: Forms and Mutations
description: Learn how to handle form submissions and data mutations with Next.js.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<PagesOnly>

Forms enable you to create and update data in web applications. Next.js provides a powerful way to handle form submissions and data mutations using **API Routes**.

> **Good to know:**
>
> - We will soon recommend [incrementally adopting](/docs/app/building-your-application/upgrading/app-router-migration) the App Router and using [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations#how-server-actions-work) for handling form submissions and data mutations. Server Actions allow you to define asynchronous server functions that can be called directly from your components, without needing to manually create an API Route.
> - API Routes [do not specify CORS headers](https://developer.mozilla.org/docs/Web/HTTP/CORS), meaning they are same-origin only by default.
> - Since API Routes run on the server, we're able to use sensitive values (like API keys) through [Environment Variables](/docs/pages/building-your-application/configuring/environment-variables) without exposing them to the client. This is critical for the security of your application.

</PagesOnly>

<AppOnly>

Forms enable you to create and update data in web applications. Next.js provides a powerful way to handle form submissions and data mutations using **Server Actions**.

<details>
  <summary>Examples</summary>

- [Form with Loading & Error States](https://github.com/vercel/next.js/tree/canary/examples/next-forms)

</details>

## How Server Actions Work

With Server Actions, you don't need to manually create an API endpoints. Instead, you define asynchronous server functions that can be called directly from your components.

> **🎥 Watch:** Learn more about forms and mutations with the App Router → [YouTube (10 minutes)](https://youtu.be/dDpZfOQBMaU?

si=cJZHlUu_jFhCzHUg).

Server Actions can be defined in Server Components or called from Client Components. Defining the action in a Server Component allows the form to function without JavaScript, providing progressive enhancement.

> **Good to know:**
>
> - Forms calling Server Actions from Server Components can function without JavaScript.
> - Forms calling Server Actions from Client Components will queue submissions if JavaScript isn't loaded yet, prioritizing client hydration.
> - Server Actions inherit the [runtime](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes) from the page or layout they are used on.
> - Server Actions work with fully static routes (including revalidating data with ISR).

## Revalidating Cached Data

Server Actions integrate deeply with the Next.js [caching and revalidation](/docs/app/building-your-application/caching) architecture. When a form is submitted, the Server Action can update cached data and revalidate any cache keys that should change.

Rather than being limited to a single form per route like traditional applications, Server Actions enable having multiple actions per route. Further, the browser does not need to refresh on form submission. In a single network roundtrip, Next.js can return both the updated UI and the refreshed data.

View the examples below for [revalidating data from Server Actions](#revalidating-data).

</AppOnly>

## Examples

### Server-only Forms

<PagesOnly>

With the Pages Router, you need to manually create API endpoints to handle securely mutating data on the server.

```ts filename="pages/api/submit.ts" switcher
import type { NextApiRequest, NextApiResponse } from 'next'
```

```
export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const data = req.body
  const id = await createItem(data)
  res.status(200).json({ id })
}
```

```js filename="pages/api/submit.js" switcher
export default function handler(req, res) {
  const data = req.body
  const id = await createItem(data)
  res.status(200).json({ id })
}
```

Then, call the API Route from the client with an event handler:

```tsx filename="pages/index.tsx" switcher
import { FormEvent } from 'react'

export default function Page() {
  async function onSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()

    const formData = new FormData(event.currentTarget)
    const response = await fetch('/api/submit', {
      method: 'POST',
      body: formData,
    })

    // Handle response if necessary
    const data = await response.json()
    // ...
  }

  return (
    <form onSubmit={onSubmit}>
      <input type="text" name="name" />
      <button type="submit">Submit</button>
    </form>
  )
}
```

```jsx filename="pages/index.jsx" switcher
export default function Page() {
  async function onSubmit(event) {
    event.preventDefault()

    const formData = new FormData(event.target)
    const response = await fetch('/api/submit', {
      method: 'POST',
      body: formData,
    })

    // Handle response if necessary
    const data = await response.json()
    // ...
  }

  return (
    <form onSubmit={onSubmit}>
      <input type="text" name="name" />
      <button type="submit">Submit</button>
    </form>
  )
}
```

</PagesOnly>

<AppOnly>

To create a server-only form, define the Server Action in a Server Component. The action can either be defined inline with the `"use server"` directive at the top of the function, or in a separate file with the directive at the top of the file.

```tsx filename="app/page.tsx" switcher
export default function Page() {
  async function create(formData: FormData) {
    'use server'

    // mutate data
    // revalidate cache
  }

  return <form action={create}>...</form>
}
```

```jsx filename="app/page.jsx" switcher
export default function Page() {
  async function create(formData) {
    'use server'

    // mutate data
    // revalidate cache
  }

  return <form action={create}>...</form>
}
```

> **Good to know**: `<form action={create}>` takes the [FormData](https://developer.mozilla.org/docs/Web/API/FormData/FormData) data type. In the example above, the FormData submitted via the HTML [`form`](https://developer.mozilla.org/docs/Web/HTML/Element/form) is accessible in the server action `create`.

### Revalidating Data

Server Actions allow you to invalidate the [Next.js Cache](/docs/app/building-your-application/caching) on demand. You can invalidate an entire route segment with [`revalidatePath`](/docs/app/api-reference/functions/revalidatePath):

```ts filename="app/actions.ts" switcher
'use server'

import { revalidatePath } from 'next/cache'

export default async function submit() {
  await submitForm()
  revalidatePath('/')
}
```

```js filename="app/actions.js" switcher
'use server'

import { revalidatePath } from 'next/cache'

export default async function submit() {
  await submitForm()
  revalidatePath('/')
}
```

Or invalidate a specific data fetch with a cache tag using [`revalidateTag`](/docs/app/api-reference/functions/revalidateTag):

```ts filename="app/actions.ts" switcher
'use server'

import { revalidateTag } from 'next/cache'

export default async function submit() {
  await addPost()
  revalidateTag('posts')
}
```

```js filename="app/actions.js" switcher
'use server'

import { revalidateTag } from 'next/cache'

export default async function submit() {
  await addPost()
  revalidateTag('posts')
}
```

</AppOnly>

### Redirecting

<PagesOnly>

If you would like to redirect the user to a different route after a mutation, you can [`redirect`](/docs/pages/building-your-application/routing/api-routes#response-helpers) to any absolute or relative URL:

```ts filename="pages/api/submit.ts" switcher
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const id = await addPost()
  res.redirect(307, `/post/${id}`)
}
```

```js filename="pages/api/submit.js" switcher
export default async function handler(req, res) {
  const id = await addPost()
  res.redirect(307, `/post/${id}`)
}
```

</PagesOnly>

<AppOnly>

If you would like to redirect the user to a different route after the completion of a Server Action, you can use [`redirect`](/docs/app/api-reference/functions/redirect) and any absolute or relative URL:

```ts filename="app/actions.ts" switcher
'use server'

import { redirect } from 'next/navigation'
import { revalidateTag } from 'next/cache'

export default async function submit() {
  const id = await addPost()
  revalidateTag('posts') // Update cached posts
  redirect(`/post/${id}`) // Navigate to new route
}
```

```js filename="app/actions.js" switcher
'use server'

import { redirect } from 'next/navigation'
import { revalidateTag } from 'next/cache'

export default async function submit() {
  const id = await addPost()
  revalidateTag('posts') // Update cached posts
  redirect(`/post/${id}`) // Navigate to new route
}
```

</AppOnly>

### Form Validation

We recommend using HTML validation like `required` and `type="email"` for basic form validation.

For more advanced server-side validation, use a schema validation library like [zod](https://zod.dev/) to validate the structure of the parsed form data:

<PagesOnly>

```ts filename="pages/api/submit.ts" switcher
import type { NextApiRequest, NextApiResponse } from 'next'
import { z } from 'zod'

const schema = z.object({
  // ...
})

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const parsed = schema.parse(req.body)
  // ...
}
```

```js filename="pages/api/submit.js" switcher
import { z } from 'zod'

const schema = z.object({
  // ...
})

export default async function handler(req, res) {
  const parsed = schema.parse(req.body)
  // ...
}
```

</PagesOnly>

<AppOnly>

```tsx filename="app/actions.ts" switcher
import { z } from 'zod'

const schema = z.object({
```

  // ...
})

export default async function submit(formData: FormData) {
  const parsed = schema.parse({
    id: formData.get('id'),
  })
  // ...
}

```jsx filename="app/actions.js" switcher
import { z } from 'zod'

const schema = z.object({
  // ...
})

export default async function submit(formData) {
  const parsed = schema.parse({
    id: formData.get('id'),
  })
  // ...
}
```

</AppOnly>

### Displaying Loading State

<AppOnly>

Use the [`useFormStatus`](https://react.dev/reference/react-dom/hooks/useFormStatus) hook to show a loading state when a form is submitting on the server. The `useFormStatus` hook can only be used as a child of a `form` element using a Server Action.

For example, the following submit button:

```tsx filename="app/submit-button.tsx" switcher
'use client'

import { useFormStatus } from 'react-dom'

export function SubmitButton() {
  const { pending } = useFormStatus()
```

~~~
  return (
    <button type="submit" aria-disabled={pending}>
      Add
    </button>
  )
}
```

```jsx filename="app/submit-button.jsx" switcher
'use client'

import { useFormStatus } from 'react-dom'

export function SubmitButton() {
  const { pending } = useFormStatus()

  return (
    <button type="submit" aria-disabled={pending}>
      Add
    </button>
  )
}
```

`<SubmitButton />` can then be used in a form with a Server Action:

```tsx filename="app/page.tsx" switcher
import { SubmitButton } from '@/app/submit-button'

export default async function Home() {
  return (
    <form action={...}>
      <input type="text" name="field-name" />
      <SubmitButton />
    </form>
  )
}
```

```jsx filename="app/page.jsx" switcher
import { SubmitButton } from '@/app/submit-button'

export default async function Home() {
  return (
    <form action={...}>
      <input type="text" name="field-name" />
~~~

~~~
      <SubmitButton />
    </form>
  )
}
```

</AppOnly>

<PagesOnly>

You can use React state to show a loading state when a form is submitting on
the server:

```tsx filename="pages/index.tsx" switcher
import React, { useState, FormEvent } from 'react'

export default function Page() {
  const [isLoading, setIsLoading] = useState<boolean>(false)

  async function onSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()
    setIsLoading(true) // Set loading to true when the request starts

    try {
      const formData = new FormData(event.currentTarget)
      const response = await fetch('/api/submit', {
        method: 'POST',
        body: formData,
      })

      // Handle response if necessary
      const data = await response.json()
      // ...
    } catch (error) {
      // Handle error if necessary
      console.error(error)
    } finally {
      setIsLoading(false) // Set loading to false when the request completes
    }
  }

  return (
    <form onSubmit={onSubmit}>
      <input type="text" name="name" />
      <button type="submit" disabled={isLoading}>
        {isLoading ? 'Loading...' : 'Submit'}
      </button>
~~~

~~~
    </form>
  )
}
```

```jsx filename="pages/index.jsx" switcher
import React, { useState } from 'react'

export default function Page() {
  const [isLoading, setIsLoading] = useState(false)

  async function onSubmit(event) {
    event.preventDefault()
    setIsLoading(true) // Set loading to true when the request starts

    try {
      const formData = new FormData(event.currentTarget)
      const response = await fetch('/api/submit', {
        method: 'POST',
        body: formData,
      })

      // Handle response if necessary
      const data = await response.json()
      // ...
    } catch (error) {
      // Handle error if necessary
      console.error(error)
    } finally {
      setIsLoading(false) // Set loading to false when the request completes
    }
  }

  return (
    <form onSubmit={onSubmit}>
      <input type="text" name="name" />
      <button type="submit" disabled={isLoading}>
        {isLoading ? 'Loading...' : 'Submit'}
      </button>
    </form>
  )
}
```

</PagesOnly>

### Error Handling
~~~

<AppOnly>

Server Actions can also return [serializable objects](https://developer.mozilla.org/docs/Glossary/Serialization). For example, your Server Action might handle errors from creating a new item:

```ts filename="app/actions.ts" switcher
'use server'

export async function createTodo(prevState: any, formData: FormData) {
  try {
    await createItem(formData.get('todo'))
    return revalidatePath('/')
  } catch (e) {
    return { message: 'Failed to create' }
  }
}
```

```js filename="app/actions.js" switcher
'use server'

export async function createTodo(prevState, formData) {
  try {
    await createItem(formData.get('todo'))
    return revalidatePath('/')
  } catch (e) {
    return { message: 'Failed to create' }
  }
}
```

Then, from a Client Component, you can read this value and display an error message.

```tsx filename="app/add-form.tsx" switcher
'use client'

import { useFormState, useFormStatus } from 'react-dom'
import { createTodo } from '@/app/actions'

const initialState = {
  message: null,
}

function SubmitButton() {
```

~~~
  const { pending } = useFormStatus()

  return (
    <button type="submit" aria-disabled={pending}>
      Add
    </button>
  )
}

export function AddForm() {
  const [state, formAction] = useFormState(createTodo, initialState)

  return (
    <form action={formAction}>
      <label htmlFor="todo">Enter Task</label>
      <input type="text" id="todo" name="todo" required />
      <SubmitButton />
      <p aria-live="polite" className="sr-only">
        {state?.message}
      </p>
    </form>
  )
}
```

```jsx filename="app/add-form.jsx" switcher
'use client'

import { useFormState, useFormStatus } from 'react-dom'
import { createTodo } from '@/app/actions'

const initialState = {
  message: null,
}

function SubmitButton() {
  const { pending } = useFormStatus()

  return (
    <button type="submit" aria-disabled={pending}>
      Add
    </button>
  )
}

export function AddForm() {
  const [state, formAction] = useFormState(createTodo, initialState)
~~~

```
  return (
    <form action={formAction}>
      <label htmlFor="todo">Enter Task</label>
      <input type="text" id="todo" name="todo" required />
      <SubmitButton />
      <p aria-live="polite" className="sr-only">
        {state?.message}
      </p>
    </form>
  )
}
```

</AppOnly>

<PagesOnly>

You can use React state to show an error message when a form submission fails:

```tsx filename="pages/index.tsx" switcher
import React, { useState, FormEvent } from 'react'

export default function Page() {
  const [isLoading, setIsLoading] = useState<boolean>(false)
  const [error, setError] = useState<string | null>(null)

  async function onSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()
    setIsLoading(true)
    setError(null) // Clear previous errors when a new request starts

    try {
      const formData = new FormData(event.currentTarget)
      const response = await fetch('/api/submit', {
        method: 'POST',
        body: formData,
      })

      if (!response.ok) {
        throw new Error('Failed to submit the data. Please try again.')
      }

      // Handle response if necessary
      const data = await response.json()
      // ...
```

~~~
    } catch (error) {
      // Capture the error message to display to the user
      setError(error.message)
      console.error(error)
    } finally {
      setIsLoading(false)
    }
  }

  return (
    <div>
      {error && <div style={{ color: 'red' }}>{error}</div>}
      <form onSubmit={onSubmit}>
        <input type="text" name="name" />
        <button type="submit" disabled={isLoading}>
          {isLoading ? 'Loading...' : 'Submit'}
        </button>
      </form>
    </div>
  )
}
```

```jsx filename="pages/index.jsx" switcher
import React, { useState } from 'react'

export default function Page() {
  const [isLoading, setIsLoading] = useState(false)
  const [error, setError] = useState(null)

  async function onSubmit(event) {
    event.preventDefault()
    setIsLoading(true)
    setError(null) // Clear previous errors when a new request starts

    try {
      const formData = new FormData(event.currentTarget)
      const response = await fetch('/api/submit', {
        method: 'POST',
        body: formData,
      })

      if (!response.ok) {
        throw new Error('Failed to submit the data. Please try again.')
      }

      // Handle response if necessary
~~~

~~~
      const data = await response.json()
      // ...
    } catch (error) {
      // Capture the error message to display to the user
      setError(error.message)
      console.error(error)
    } finally {
      setIsLoading(false)
    }
  }

  return (
    <div>
      {error && <div style={{ color: 'red' }}>{error}</div>}
      <form onSubmit={onSubmit}>
        <input type="text" name="name" />
        <button type="submit" disabled={isLoading}>
          {isLoading ? 'Loading...' : 'Submit'}
        </button>
      </form>
    </div>
  )
}
```

</PagesOnly>

<AppOnly>

### Optimistic Updates

Use [`useOptimistic`](https://react.dev/reference/react/useOptimistic) to
optimistically update the UI before the Server Action finishes, rather than
waiting for the response:

```tsx filename="app/page.tsx" switcher
'use client'

import { useOptimistic } from 'react'
import { send } from './actions'

type Message = {
  message: string
}

export function Thread({ messages }: { messages: Message[] }) {
  const [optimisticMessages, addOptimisticMessage] =
~~~

```tsx
useOptimistic<Message[]>(
    messages,
    (state: Message[], newMessage: string) => [
      ...state,
      { message: newMessage },
    ]
  )

  return (
    <div>
      {optimisticMessages.map((m, k) => (
        <div key={k}>{m.message}</div>
      ))}
      <form
        action={async (formData: FormData) => {
          const message = formData.get('message')
          addOptimisticMessage(message)
          await send(message)
        }}
      >
        <input type="text" name="message" />
        <button type="submit">Send</button>
      </form>
    </div>
  )
}
```

```jsx filename="app/page.jsx" switcher
'use client'

import { useOptimistic } from 'react'
import { send } from './actions'

export function Thread({ messages }) {
  const [optimisticMessages, addOptimisticMessage] = useOptimistic(
    messages,
    (state, newMessage) => [...state, { message: newMessage }]
  )

  return (
    <div>
      {optimisticMessages.map((m) => (
        <div>{m.message}</div>
      ))}
      <form
        action={async (formData) => {
```

          const message = formData.get('message')
          addOptimisticMessage(message)
          await send(message)
        }}
      >
        <input type="text" name="message" />
        <button type="submit">Send</button>
      </form>
    </div>
)
}

</AppOnly>

### Setting Cookies

<PagesOnly>

You can set cookies inside an API Route using the `setHeader` method on the response:

```ts filename="pages/api/cookie.ts" switcher
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  res.setHeader('Set-Cookie', 'username=lee; Path=/; HttpOnly')
  res.status(200).send('Cookie has been set.')
}
```

```js filename="pages/api/cookie.js" switcher
export default async function handler(req, res) {
  res.setHeader('Set-Cookie', 'username=lee; Path=/; HttpOnly')
  res.status(200).send('Cookie has been set.')
}
```

</PagesOnly>

<AppOnly>

You can set cookies inside a Server Action using the [`cookies`](/docs/app/api-reference/functions/cookies) function:

```ts filename="app/actions.ts" switcher
'use server'

import { cookies } from 'next/headers'

export async function create() {
  const cart = await createCart()
  cookies().set('cartId', cart.id)
}
```

```js filename="app/actions.js" switcher
'use server'

import { cookies } from 'next/headers'

export async function create() {
  const cart = await createCart()
  cookies().set('cartId', cart.id)
}
```

</AppOnly>

### Reading Cookies

<PagesOnly>

You can read cookies inside an API Route using the [`cookies`](/docs/pages/building-your-application/routing/api-routes#request-helpers) request helper:

```ts filename="pages/api/cookie.ts" switcher
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const auth = req.cookies.authorization
  // ...
}
```

```js filename="pages/api/cookie.js" switcher
export default async function handler(req, res) {
  const auth = req.cookies.authorization
```

  // ...
}

</PagesOnly>

<AppOnly>

You can read cookies inside a Server Action using the [`cookies`](/docs/app/api-reference/functions/cookies) function:

```ts filename="app/actions.ts" switcher
'use server'

import { cookies } from 'next/headers'

export async function read() {
  const auth = cookies().get('authorization')?.value
  // ...
}
```

```js filename="app/actions.js" switcher
'use server'

import { cookies } from 'next/headers'

export async function read() {
  const auth = cookies().get('authorization')?.value
  // ...
}
```

</AppOnly>

### Deleting Cookies

<PagesOnly>

You can delete cookies inside an API Route using the `setHeader` method on the response:

```ts filename="pages/api/cookie.ts" switcher
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
```

  res: NextApiResponse
) {
  res.setHeader('Set-Cookie', 'username=; Path=/; HttpOnly; Max-Age=0')
  res.status(200).send('Cookie has been deleted.')
}

```js filename="pages/api/cookie.js" switcher
export default async function handler(req, res) {
  res.setHeader('Set-Cookie', 'username=; Path=/; HttpOnly; Max-Age=0')
  res.status(200).send('Cookie has been deleted.')
}
```

</PagesOnly>

<AppOnly>

You can delete cookies inside a Server Action using the [`cookies`](/docs/app/api-reference/functions/cookies) function:

```ts filename="app/actions.ts" switcher
'use server'

import { cookies } from 'next/headers'

export async function delete() {
  cookies().delete('name')
  // ...
}
```

```js filename="app/actions.js" switcher
'use server'

import { cookies } from 'next/headers'

export async function delete() {
  cookies().delete('name')
  // ...
}
```

See [additional examples](/docs/app/api-reference/functions/cookies#deleting-cookies) for deleting cookies from Server Actions.

</AppOnly>

```
---
title: Server Components
description: Learn how you can use React Server Components to render parts
of your application on the server.
related:
  description: Learn how Next.js caches data and the result of static rendering.
  links:
    - app/building-your-application/caching
---
```

React Server Components allow you to write UI that can be rendered and optionally cached on the server. In Next.js, the rendering work is further split by route segments to enable streaming and partial rendering, and there are three different server rendering strategies:

- [Static Rendering](#static-rendering-default)
- [Dynamic Rendering](#dynamic-rendering)
- [Streaming](#streaming)

This page will go through how Server Components work, when you might use them, and the different server rendering strategies.

## Benefits of Server Rendering

There are a couple of benefits to doing the rendering work on the server, including:

- **Data Fetching**: Server Components allow you to move data fetching to the server, closer to your data source. This can improve performance by reducing time it takes to fetch data needed for rendering, and the amount of requests the client needs to make.
- **Security**: Server Components allow you to keep sensitive data and logic on the server, such as tokens and API keys, without the risk of exposing them to the client.
- **Caching**: By rendering on the server, the result can be cached and reused on subsequent requests and across users. This can improve performance and reduce cost by reducing the amount of rendering and data fetching done on each request.
- **Bundle Sizes**: Server Components allow you to keep large dependencies that previously would impact the client JavaScript bundle size on the server. This is beneficial for users with slower internet or less powerful devices, as the client does not have to download, parse and execute any JavaScript for Server Components.
- **Initial Page Load and [First Contentful Paint (FCP)](https://web.dev/fcp/)**: On the server, we can generate HTML to allow users to view the page immediately, without waiting for the client to download, parse and execute the

JavaScript needed to render the page.
- **Search Engine Optimization and Social Network Shareability**: The rendered HTML can be used by search engine bots to index your pages and social network bots to generate social card previews for your pages.
- **Streaming**: Server Components allow you to split the rendering work into chunks and stream them to the client as they become ready. This allows the user to see parts of the page earlier without having to wait for the entire page to be rendered on the server.

## Using Server Components in Next.js

By default, Next.js uses Server Components. This allows you to automatically implement server rendering with no additional configuration, and you can opt into using Client Components when needed, see [Client Components](/docs/app/building-your-application/rendering/client-components).

## How are Server Components rendered?

On the server, Next.js uses React's APIs to orchestrate rendering. The rendering work is split into chunks: by individual route segments and [Suspense Boundaries](https://react.dev/reference/react/Suspense).

Each chunk is rendered in two steps:

1. React renders Server Components into a special data format called the **React Server Component Payload (RSC Payload)**.
2. Next.js uses the RSC Payload and Client Component JavaScript instructions to render **HTML** on the server.

{/* Rendering Diagram */}

Then, on the client:

1. The HTML is used to immediately show a fast non-interactive preview of the route - this is for the initial page load only.
2. The React Server Components Payload is used to reconcile the Client and Server Component trees, and update the DOM.
3. The JavaScript instructions are used to [hydrate](https://react.dev/reference/react-dom/client/hydrateRoot) Client Components and make the application interactive.

> **What is the React Server Component Payload (RSC)?**
>
> The RSC Payload is a compact binary representation of the rendered React Server Components tree. It's used by React on the client to update the browser's DOM. The RSC Payload contains:
>

> - The rendered result of Server Components
> - Placeholders for where Client Components should be rendered and references to their JavaScript files
> - Any props passed from a Server Component to a Client Component

## Server Rendering Strategies

There are three subsets of server rendering: Static, Dynamic, and Streaming.

### Static Rendering (Default)

{/* Static Rendering Diagram */}

With Static Rendering, routes are rendered at **build time**, or in the background after [data revalidation](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#revalidating-data). The result is cached and can be pushed to a [Content Delivery Network (CDN)](https://developer.mozilla.org/docs/Glossary/CDN). This optimization allows you to share the result of the rendering work between users and server requests.

Static rendering is useful when a route has data that is not personalized to the user and can be known at build time, such as a static blog post or a product page.

### Dynamic Rendering

{/* Dynamic Rendering Diagram */}

With Dynamic Rendering, routes are rendered for each user at **request time**.

Dynamic rendering is useful when a route has data that is personalized to the user or has information that can only be known at request time, such as cookies or the URL's search params.

> **Dynamic Routes with Cached Data**
>
> In most websites, routes are not fully static or fully dynamic - it's a spectrum. For example, you can have an e-commerce page that uses cached product data that's revalidated at an interval, but also has uncached, personalized customer data.
>
> In Next.js, you can have dynamically rendered routes that have both cached and uncached data. This is because the RSC Payload and data are cached separately. This allows you to opt into dynamic rendering without worrying about the performance impact of fetching all the data at request time.
>

> Learn more about the [full-route cache](/docs/app/building-your-application/caching#full-route-cache) and [Data Cache](/docs/app/building-your-application/caching#data-cache).

#### Switching to Dynamic Rendering

During rendering, if a [dynamic function](#dynamic-functions) or [uncached data request](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#opting-out-of-data-caching) is discovered, Next.js will switch to dynamically rendering the whole route. This table summarizes how dynamic functions and data caching affect whether a route is statically or dynamically rendered:

Dynamic Functions	Data	Route
No	Cached	Statically Rendered
Yes	Cached	Dynamically Rendered
No	Not Cached	Dynamically Rendered
Yes	Not Cached	Dynamically Rendered

In the table above, for a route to be fully static, all data must be cached. However, you can have a dynamically rendered route that uses both cached and uncached data fetches.

As a developer, you do not need to choose between static and dynamic rendering as Next.js will automatically choose the best rendering strategy for each route based on the features and APIs used. Instead, you choose when to [cache or revalidate specific data](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating), and you may choose to [stream](#streaming) parts of your UI.

#### 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:

- **[`cookies()`](/docs/app/api-reference/functions/cookies) and [`headers()`](/docs/app/api-reference/functions/headers)**: Using these in a Server Component will opt the whole route into dynamic rendering at request time.
- **[`useSearchParams()`](/docs/app/api-reference/functions/use-search-params)**:
  - In Client Components, it'll 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](/docs/app/api-reference/functions/use-search-params#static-rendering).
- **[`searchParams`](/docs/app/api-reference/file-conventions/page#searchparams-optional)**: Using the [Pages](/docs/app/api-reference/file-conventions/page) prop will opt the page into dynamic rendering at request time.

Using any of these functions will opt the whole route into dynamic rendering at request time.

### Streaming

<Image
  alt="Diagram showing parallelization of route segments during streaming, showing data fetching, rendering, and hydration of invidiual chunks."
  srcLight="/docs/light/sequential-parallel-data-fetching.png"
  srcDark="/docs/dark/sequential-parallel-data-fetching.png"
  width="1600"
  height="525"
/>

Streaming enables you to progressively render UI from the server. Work is split into chunks and streamed to the client as it becomes ready. This allows the user to see parts of the page immediately, before the entire content has finished rendering.

<Image
  alt="Diagram showing partially rendered page on the client, with loading UI for chunks that are being streamed."
  srcLight="/docs/light/server-rendering-with-streaming.png"
  srcDark="/docs/dark/server-rendering-with-streaming.png"
  width="1600"
  height="785"
/>

Streaming is built into the Next.js App Router by default. This helps improve both the initial page loading performance, as well as UI that depends on slower data fetches that would block rendering the whole route. For example, reviews on a product page.

You can start streaming route segments using `loading.js` and UI components with [React Suspense](/docs/app/building-your-application/routing/loading-ui-and-streaming). See the [Loading UI and Streaming](/docs/app/building-your-application/routing/loading-ui-and-streaming) section for more information.

---
title: Client Components

description: Learn how to use Client Components to render parts of your application on the client.

Client Components allows you to write interactive UI that can be rendered on the client at request time. In Next.js, client rendering is **opt-in**, meaning you have to explicitly decide what components React should render on the client.

This page will go through how Client Components work, how they're rendered, and when you might use them.

## Benefits of Client Rendering

There are a couple of benefits to doing the rendering work on the client, including:

- **Interactivity**: Client Components can use state, effects, and event listeners, meaning they can provide immediate feedback to the user and update the UI.
- **Browser APIs**: Client Components have access to browser APIs, like [geolocation](https://developer.mozilla.org/docs/Web/API/Geolocation_API) or [localStorage](https://developer.mozilla.org/docs/Web/API/Window/localStorage), allowing you to build UI for specific use cases.

## Using Client Components in Next.js

To use Client Components, you can add the React [`"use client"` directive](https://react.dev/reference/react/use-client) at the top of a file, above your imports.

`"use client"` is used to declare a [boundary](/docs/app/building-your-application/rendering#network-boundary) between a Server and Client Component modules. This means that by defining a `"use client"` in a file, all other modules imported into it, including child components, are considered part of the client bundle.

```tsx filename="app/counter.tsx" highlight={1} switcher
'use client'

import { useState } from 'react'

export default function Counter() {
  const [count, setCount] = useState(0)

  return (
    <div>
```

```
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

```jsx filename="app/counter.js" highlight={1} switcher
'use client'

import { useState } from 'react'

export default function Counter() {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  )
}
```

The diagram below shows that using `onClick` and `useState` in a nested component (`toggle.js`) will cause an error if the `"use client"` directive is not defined. This is because, by default, the components are rendered on the server where these APIs are not available. By defining the `"use client"` directive in `toggle.js`, you can tell React to render the component and its children on the client, where the APIs are available.

<Image
  alt="Use Client Directive and Network Boundary"
  srcLight="/docs/light/use-client-directive.png"
  srcDark="/docs/dark/use-client-directive.png"
  width="1600"
  height="1320"
/>

> **Defining multiple `use client` entry points**:
>
> You can define multiple "use client" entry points in your React Component tree. This allows you to split your application into multiple client bundles (or branches).
>
> However, `"use client"` doesn't need to be defined in every component that needs to be rendered on the client. Once you define the boundary, all child

components and modules imported into it are considered part of the client bundle.

## How are Client Components Rendered?

In Next.js, Client Components are rendered differently depending on whether the request is part of a full page load (an initial visit to your application or a page reload triggered by a browser refresh) or a subsequent navigation.

### Full page load

To optimize the initial page load, Next.js will use React's APIs to render a static HTML preview on the server for both Client and Server Components. This means, when the user first visits your application, they will see the content of the page immediately, without having to wait for the client to download, parse, and execute the Client Component JavaScript bundle.

On the server:

1. React renders Server Components into a special data format called the **React Server Component Payload (RSC Payload)**, which includes references to Client Components.
2. Next.js uses the RSC Payload and Client Component JavaScript instructions to render **HTML** for the route on the server.

Then, on the client:

1. The HTML is used to immediately show a fast non-interactive initial preview of the route.
2. The React Server Components Payload is used to reconcile the Client and Server Component trees, and update the DOM.
3. The JavaScript instructions are used to [hydrate](https://react.dev/reference/react-dom/client/hydrateRoot) Client Components and make their UI interactive.

> **What is hydration?**
>
> Hydration is the process of attaching event listeners to the DOM, to make the static HTML interactive. Behind the scenes, hydration is done with the [`hydrateRoot`](https://react.dev/reference/react-dom/client/hydrateRoot) React API.

### Subsequent Navigations

On subsequent navigations, Client Components are rendered entirely on the client, without the server-rendered HTML.

This means the Client Component JavaScript bundle is downloaded and parsed. Once the bundle is ready, React will use the RSC Payload to reconcile the Client and Server Component trees, and update the DOM.

## Going back to the Server Environment

Sometimes, after you've declared the `"use client"` boundary, you may want to go back to the server environment. For example, you may want to reduce the client bundle size, fetch data on the server, or use an API that is only available on the server.

You can keep code on the server even though it's theoretically nested inside Client Components by interleaving Client and Server Components and [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations). See the [Composition Patterns](/docs/app/building-your-application/rendering/composition-patterns) page for more information.

---
title: Server and Client Composition Patterns
nav_title: Composition Patterns
description: Recommended patterns for using Server and Client Components.

When building React applications, you will need to consider what parts of your application should be rendered on the server or the client. This page covers some recommended composition patterns when using Server and Client Components.

## When to use Server and Client Components?

Here's a quick summary of the different use cases for Server and Client Components:

What do you need to do?	Server Component	Client Component
Fetch data	<Check size={18} />	<Cross size={18} />
Access backend resources (directly)	<Check size={18} />	<Cross size={18} />
Keep sensitive information on the server (access tokens, API keys, etc)	<Check size={18} />	<Cross size={18} />
Keep large dependencies on the server / Reduce client-side JavaScript	<Check size={18} />	<Cross size={18} />

Add interactivity and event listeners (`onClick()`, `onChange()`, etc)	<Cross size={18} />	<Check size={18} />
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)	<Cross size={18} />	<Check size={18} />
Use browser-only APIs	<Cross size={18} />	<Check size={18} />
Use custom hooks that depend on state, effects, or browser-only APIs	<Cross size={18} />	<Check size={18} />
Use [React Class components](https://react.dev/reference/react/Component)	<Cross size={18} />	<Check size={18} />

## Server Component Patterns

Before opting into client-side rendering, you may wish to do some work on the server like fetching data, or accessing your database or backend services.

Here are some common patterns when working with Server Components:

### Sharing data between components

When fetching data on the server, there may be cases where you need to share data across different components. For example, you may have a layout and a page that depend on the same data.

Instead of using [React Context](https://react.dev/learn/passing-data-deeply-with-context) (which is not available on the server) or passing data as props, you can use [`fetch`](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#fetching-data-on-the-server-with-fetch) or React's `cache` function to fetch the same data in the components that need it, without worrying about making duplicate requests for the same data. This is because React extends `fetch` to automatically memoize data requests, and the `cache` function can be used when `fetch` is not available.

Learn more about [memoization](/docs/app/building-your-application/caching#request-memoization) in React.

### Keeping Server-only Code out of the Client Environment

Since JavaScript modules can be shared between both Server and Client Components modules, it's possible for code that was only ever intended to be run on the server to sneak its way into the client.

For example, take the following data-fetching function:

```ts filename="lib/data.ts" switcher
export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    headers: {
      authorization: process.env.API_KEY,
    },
  })

  return res.json()
}
```

```js filename="lib/data.js" switcher
export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    headers: {
      authorization: process.env.API_KEY,
    },
  })

  return res.json()
}
```

At first glance, it appears that `getData` works on both the server and the client. However, this function contains an `API_KEY`, written with the intention that it would only ever be executed on the server.

Since the environment variable `API_KEY` is not prefixed with `NEXT_PUBLIC`, it's a private variable that can only be accessed on the server. To prevent your environment variables from being leaked to the client, Next.js replaces private environment variables with an empty string.

As a result, even though `getData()` can be imported and executed on the client, it won't work as expected. And while making the variable public would make the function work on the client, you may not want to expose sensitive information to the client.

To prevent this sort of unintended client usage of server code, we can use the `server-only` package to give other developers a build-time error if they ever accidentally import one of these modules into a Client Component.

To use `server-only`, first install the package:

```bash filename="Terminal"
npm install server-only
```

Then import the package into any module that contains server-only code:

```js filename="lib/data.js"
import 'server-only'

export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    headers: {
      authorization: process.env.API_KEY,
    },
  })

  return res.json()
}
```

Now, any Client Component that imports `getData()` will receive a build-time error explaining that this module can only be used on the server.

The corresponding package `client-only` can be used to mark modules that contain client-only code – for example, code that accesses the `window` object.

### Using Third-party Packages and Providers

Since Server Components are a new React feature, third-party packages and providers in the ecosystem are just beginning to add the `"use client"` directive to components that use client-only features like `useState`, `useEffect`, and `createContext`.

Today, many components from `npm` packages that use client-only features do not yet have the directive. These third-party components will work as expected within Client Components since they have the `"use client"` directive, but they won't work within Server Components.

For example, let's say you've installed the hypothetical `acme-carousel` package which has a `<Carousel />` component. This component uses `useState`, but it doesn't yet have the `"use client"` directive.

If you use `<Carousel />` within a Client Component, it will work as expected:

```tsx filename="app/gallery.tsx" switcher
'use client'

import { useState } from 'react'
import { Carousel } from 'acme-carousel'

export default function Gallery() {
  let [isOpen, setIsOpen] = useState(false)

  return (
    <div>
      <button onClick={() => setIsOpen(true)}>View pictures</button>

      {/* Works, since Carousel is used within a Client Component */}
      {isOpen && <Carousel />}
    </div>
  )
}
```

```jsx filename="app/gallery.js" switcher
'use client'

import { useState } from 'react'
import { Carousel } from 'acme-carousel'

export default function Gallery() {
  let [isOpen, setIsOpen] = useState(false)

  return (
    <div>
      <button onClick={() => setIsOpen(true)}>View pictures</button>

      {/*  Works, since Carousel is used within a Client Component */}
      {isOpen && <Carousel />}
    </div>
  )
}
```

However, if you try to use it directly within a Server Component, you'll see an error:

```tsx filename="app/page.tsx" switcher
import { Carousel } from 'acme-carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>

      {/* Error: `useState` can not be used within Server Components */}
      <Carousel />
    </div>
  )
}
```

```jsx filename="app/page.js" switcher
import { Carousel } from 'acme-carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>

      {/*  Error: `useState` can not be used within Server Components */}
      <Carousel />
    </div>
  )
}
```

This is because Next.js doesn't know `<Carousel />` is using client-only features.

To fix this, you can wrap third-party components that rely on client-only features in your own Client Components:

```tsx filename="app/carousel.tsx" switcher
'use client'

import { Carousel } from 'acme-carousel'

export default Carousel
```

```jsx filename="app/carousel.js" switcher
'use client'

import { Carousel } from 'acme-carousel'

export default Carousel
```

Now, you can use `<Carousel />` directly within a Server Component:

```tsx filename="app/page.tsx" switcher
import Carousel from './carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>

      {/*  Works, since Carousel is a Client Component */}
      <Carousel />
    </div>
  )
}
```

```jsx filename="app/page.js" switcher
import Carousel from './carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>

      {/*  Works, since Carousel is a Client Component */}
      <Carousel />
    </div>
  )
}
```

We don't expect you to need to wrap most third-party components since it's likely you'll be using them within Client Components. However, one exception is providers, since they rely on React state and context, and are typically needed at the root of an application. [Learn more about third-party context providers below](#using-context-providers).

#### Using Context Providers

Context providers are typically rendered near the root of an application to share global concerns, like the current theme. Since [React context](https://react.dev/learn/passing-data-deeply-with-context) is not supported in Server Components, trying to create a context at the root of your application will cause an error:

```tsx filename="app/layout.tsx" switcher
import { createContext } from 'react'

//  createContext is not supported in Server Components
export const ThemeContext = createContext({})

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <ThemeContext.Provider value="dark">{children}</
ThemeContext.Provider>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import { createContext } from 'react'

//  createContext is not supported in Server Components
export const ThemeContext = createContext({})

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <ThemeContext.Provider value="dark">{children}</
ThemeContext.Provider>
      </body>
    </html>
  )
}
```

To fix this, create your context and render its provider inside of a Client Component:

```tsx filename="app/theme-provider.tsx" switcher
'use client'

import { createContext } from 'react'

export const ThemeContext = createContext({})

export default function ThemeProvider({ children }) {
  return <ThemeContext.Provider value="dark">{children}</
ThemeContext.Provider>
}
```

```jsx filename="app/theme-provider.js" switcher
'use client'
```

import { createContext } from 'react'

export const ThemeContext = createContext({})

export default function ThemeProvider({ children }) {
  return <ThemeContext.Provider value="dark">{children}</
ThemeContext.Provider>
}

Your Server Component will now be able to directly render your provider since it's been marked as a Client Component:

```tsx filename="app/layout.tsx" switcher
import ThemeProvider from './theme-provider'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <ThemeProvider>{children}</ThemeProvider>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import ThemeProvider from './theme-provider'

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <ThemeProvider>{children}</ThemeProvider>
      </body>
    </html>
  )
}
```

With the provider rendered at the root, all other Client Components throughout your app will be able to consume this context.

> **Good to know**: You should render providers as deep as possible in the tree – notice how `ThemeProvider` only wraps `{children}` instead of the entire `<html>` document. This makes it easier for Next.js to optimize the static parts of your Server Components.

#### Advice for Library Authors

In a similar fashion, library authors creating packages to be consumed by other developers can use the `"use client"` directive to mark client entry points of their package. This allows users of the package to import package components directly into their Server Components without having to create a wrapping boundary.

You can optimize your package by using ['use client' deeper in the tree](#moving-client-components-down-the-tree), allowing the imported modules to be part of the Server Component module graph.

It's worth noting some bundlers might strip out `"use client"` directives. You can find an example of how to configure esbuild to include the `"use client"` directive in the [React Wrap Balancer](https://github.com/shuding/react-wrap-balancer/blob/main/tsup.config.ts#L10-L13) and [Vercel Analytics](https://github.com/vercel/analytics/blob/main/packages/web/tsup.config.js#L26-L30) repositories.

## Client Components

### Moving Client Components Down the Tree

To reduce the Client JavaScript bundle size, we recommend moving Client Components down your component tree.

For example, you may have a Layout that has static elements (e.g. logo, links, etc) and an interactive search bar that uses state.

Instead of making the whole layout a Client Component, move the interactive logic to a Client Component (e.g. `<SearchBar />`) and keep your layout as a Server Component. This means you don't have to send all the component Javascript of the layout to the client.

```tsx filename="app/layout.tsx" switcher
// SearchBar is a Client Component
import SearchBar from './searchbar'
// Logo is a Server Component
import Logo from './logo'
```

```
// Layout is a Server Component by default
export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <>
      <nav>
        <Logo />
        <SearchBar />
      </nav>
      <main>{children}</main>
    </>
  )
}
```

```jsx filename="app/layout.js" switcher
// SearchBar is a Client Component
import SearchBar from './searchbar'
// Logo is a Server Component
import Logo from './logo'

// Layout is a Server Component by default
export default function Layout({ children }) {
  return (
    <>
      <nav>
        <Logo />
        <SearchBar />
      </nav>
      <main>{children}</main>
    </>
  )
}
```

### Passing props from Server to Client Components (Serialization)

If you fetch data in a Server Component, you may want to pass data down as props to Client Components. Props passed from the Server to Client Components need to be [serializable](https://developer.mozilla.org/docs/Glossary/Serialization) by React.

If your Client Components depend on data that is not serializable, you can [fetch data on the client with a third party library](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#fetching-data-on-the-client-with-third-party-libraries) or on the server via a [Route Handler](/docs/app/building-your-application/routing/route-handlers).

## Interleaving Server and Client Components

When interleaving Client and Server Components, it may be helpful to visualize your UI as a tree of components. Starting with the [root layout](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required), which is a Server Component, you can then render certain subtrees of components on the client by adding the `"use client"` directive.

{/* Diagram - interleaving */}

Within those client subtrees, you can still nest Server Components or call Server Actions, however there are some things to keep in mind:

- During a request-response lifecycle, your code moves from the server to the client. If you need to access data or resources on the server while on the client, you'll be making a **new** request to the server - not switching back and forth.
- When a new request is made to the server, all Server Components are rendered first, including those nested inside Client Components. The rendered result (RSC Payload) will contain references to the locations of Client Components. Then, on the client, React uses the RSC Payload to reconcile Server and Client Components into a single tree.

{/* Diagram */}

- Since Client Components are rendered after Server Components, you cannot import a Server Component into a Client Component module (since it would require a new request back to the server). Instead, you can pass a Server Component as `props` to a Client Component. See the [unsupported pattern](#unsupported-pattern-importing-server-components-into-client-components) and [supported pattern](#supported-pattern-passing-server-components-to-client-components-as-props) sections below.

### Unsupported Pattern: Importing Server Components into Client Components

The following pattern is not supported. You cannot import a Server Component into a Client Component:

```tsx filename="app/client-component.tsx" switcher highlight={4,17}
'use client'

// You cannot import a Server Component into a Client Component.
import ServerComponent from './Server-Component'

export default function ClientComponent({
  children,
}: {
  children: React.ReactNode
}) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>

      <ServerComponent />
    </>
  )
}
```

```jsx filename="app/client-component.js" switcher highlight={3,13}
'use client'

// You cannot import a Server Component into a Client Component.
import ServerComponent from './Server-Component'

export default function ClientComponent({ children }) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>

      <ServerComponent />
    </>
  )
}
```

### Supported Pattern: Passing Server Components to Client Components as Props

The following pattern is supported. You can pass Server Components as a prop to a Client Component.

A common pattern is to use the React `children` prop to create a _"slot"_ in your Client Component.

In the example below, `<ClientComponent>` accepts a `children` prop:

```tsx filename="app/client-component.tsx" switcher highlight={6,15}
```

'use client'

import { useState } from 'react'

export default function ClientComponent({
  children,
}: {
  children: React.ReactNode
}) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>
      {children}
    </>
)
}

```jsx filename="app/client-component.js" switcher highlight={5,12}
'use client'

import { useState } from 'react'

export default function ClientComponent({ children }) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>

      {children}
    </>
  )
}
```

`<ClientComponent>` doesn't know that `children` will eventually be filled in by the result of a Server Component. The only responsibility `<ClientComponent>` has is to decide **where** `children` will eventually be placed.

In a parent Server Component, you can import both the `<ClientComponent>` and `<ServerComponent>` and pass `<ServerComponent>` as a child of `<ClientComponent>`:

```tsx filename="app/page.tsx"  highlight={11} switcher
// This pattern works:
// You can pass a Server Component as a child or prop of a
// Client Component.
import ClientComponent from './client-component'
import ServerComponent from './server-component'

// Pages in Next.js are Server Components by default
export default function Page() {
  return (
    <ClientComponent>
      <ServerComponent />
    </ClientComponent>
  )
}
```

```jsx filename="app/page.js" highlight={11} switcher
// This pattern works:
// You can pass a Server Component as a child or prop of a
// Client Component.
import ClientComponent from './client-component'
import ServerComponent from './server-component'

// Pages in Next.js are Server Components by default
export default function Page() {
  return (
    <ClientComponent>
      <ServerComponent />
    </ClientComponent>
  )
}
```

With this approach, `<ClientComponent>` and `<ServerComponent>` are decoupled and can be rendered independently. In this case, the child `<ServerComponent>` can be rendered on the server, well before `<ClientComponent>` is rendered on the client.

> **Good to know:**
>
> - The pattern of "lifting content up" has been used to avoid re-rendering a nested child component when a parent component re-renders.
> - You're not limited to the `children` prop. You can use any prop to pass JSX.

---

title: Edge and Node.js Runtimes
description: Learn about the switchable runtimes (Edge and Node.js) in Next.js.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

In the context of Next.js, runtime refers to the set of libraries, APIs, and general functionality available to your code during execution.

On the server, there are two runtimes where parts of your application code 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](/docs/app/api-reference/edge).

## 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	/	Normal	Low
[HTTP Streaming](/docs/app/building-your-application/routing/loading-ui-and-streaming)	Yes	Yes	Yes
IO	All	All	`fetch`
Scalability	/	High	Highest
Security	Normal	High	High
Latency	Normal	Low	Lowest
npm Packages	All	All	A smaller subset
[Static Rendering](/docs/app/building-your-application/rendering/server-			

components#static-rendering-default) | Yes | Yes | No
|
| [Dynamic Rendering](/docs/app/building-your-application/rendering/server-components#dynamic-rendering) | Yes | Yes | Yes
|
| [Data Revalidation w/ `fetch`](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating#revalidating-data) | Yes | Yes
| Yes |

### 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](https://vercel.com/docs/concepts/limits/overview#edge-middleware-and-edge-functions-size), 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](https://vercel.com/docs/concepts/limits/overview#serverless-function-size) including imported packages, fonts, and files.

The downside compared to routes using the [Edge](https://vercel.com/docs/concepts/functions/edge-functions) 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".

<AppOnly>

## 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](/docs/app/api-reference/file-conventions/route-segment-config). 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'`:

```tsx filename="app/page.tsx" switcher
export const runtime = 'edge' // 'nodejs' (default) | 'edge'
```

```jsx filename="app/page.js" switcher
export const runtime = 'edge' // 'nodejs' (default) | 'edge'
```

You can also define `runtime` on a layout level, which will make all routes under the layout run on the edge runtime:

```tsx filename="app/layout.tsx" switcher
export const runtime = 'edge' // 'nodejs' (default) | 'edge'
```

```jsx filename="app/layout.js" switcher
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.

</AppOnly>

> Please refer to the [Node.js Docs](https://nodejs.org/docs/latest/api/) and [Edge Docs](/docs/app/api-reference/edge) for the full list of available APIs. Both runtimes can also support [streaming](/docs/app/building-your-application/routing/loading-ui-and-streaming) depending on your deployment infrastructure.

---

title: Rendering
description: Learn the differences between Next.js rendering environments, strategies, and runtimes.

Rendering converts the code you write into user interfaces. React and Next.js allow you to create hybrid web applications where parts of your code can be rendered on the server or the client. This section will help you understand the differences between these rendering environments, strategies, and runtimes.

## Fundamentals

To start, it's helpful to be familiar with three foundational web concepts:

- The [Environments](#rendering-environments) your application code can be executed in: the server and the client.
- The [Request-Response Lifecycle](#request-response-lifecycle) that's initiated when a user visits or interacts with your application.
- The [Network Boundary](#network-boundary) that separates server and client code.

### Rendering Environments

There are two environments where web applications can be rendered: the client and the server.

```
<Image
  alt="Client and Server Environments"
  srcLight="/docs/light/client-and-server-environments.png"
  srcDark="/docs/dark/client-and-server-environments.png"
  width="1600"
  height="672"
/>
```

- 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 a user interface.
- The **server** refers to the computer in a data center that stores your application code, receives requests from a client, and sends back an appropriate response.

Historically, developers had to use different languages (e.g. JavaScript, PHP) and frameworks when writing code for the server and the client. With React, developers can use the **same language** (JavaScript), and the **same framework** (e.g. Next.js or your framework of choice). This flexibility allows you to seamlessly write code for both environments without context switching.

However, each environment has its own set of capabilities and constraints. Therefore, the code you write for the server and the client is not always the same. There are certain operations (e.g. data fetching or managing user state) that are better suited for one environment over the other.

Understanding these differences is key to effectively using React and Next.js. We'll cover the differences and use cases in more detail on the [Server](/docs/app/building-your-application/rendering/server-components) and [Client](/docs/app/building-your-application/rendering/client-components) Components pages, for now, let's continue building on our foundation.

### Request-Response Lifecycle

Broadly speaking, all websites follow the same **Request-Response Lifecycle**:

1. **User Action:** The user interacts with a web application. This could be clicking a link, submitting a form, or typing a URL directly into the browser's address bar.
2. **HTTP Request:** The client sends an [HTTP](https://developer.mozilla.org/docs/Web/HTTP) request to the server that contains necessary information about what resources are being requested, what method is being used (e.g. `GET`, `POST`), and additional data if necessary.
3. **Server:** The server processes the request and responds with the appropriate resources. This process may take a couple of steps like routing, fetching data, etc.
4. **HTTP Response:** After processing the request, the server sends an HTTP response back to the client. This response contains a status code (which tells the client whether the request was successful or not) and requested resources (e.g. HTML, CSS, JavaScript, static assets, etc).
5. **Client:** The client parses the resources to render the user interface.
6. **User Action:** Once the user interface is rendered, the user can interact with it, and the whole process starts again.

A major part of building a hybrid web application is deciding how to split the work in the lifecycle, and where to place the Network Boundary.

### Network Boundary

In web development, the **Network Boundary** is a conceptual line that separates the different environments. For example, the client and the server, or the server and the data store.

{/* Diagram: Network Boundary */}

In React, you choose where to place the client-server network boundary wherever it makes the most sense.

Behind the scenes, the work is split into two parts: the **client module graph** and the **server module graph**. The server module graph contains all the components that are rendered on the server, and the client module graph contains all components that are rendered on the client.

{/* Diagram: Client and Server Module Graphs */}

It may be helpful to think about module graphs as a visual representation of how files in your application depend on each other.

{/* For example, if you have a file called `Page.jsx` that imports a file called `Button.jsx` on the server, the module graph would look something like this: - Diagram - */}

You can use the React `"use client"` convention to define the boundary. There's also a `"use server"` convention, which tells React to do some computational work on the server.

## Building Hybrid Applications

When working in these environments, it's helpful to think of the flow of the code in your application as **unidirectional**. In other words, during a response, your application code flows in one direction: from the server to the client.

{/* Diagram: Response flow */}

If you need to access the server from the client, you send a **new** request to the server rather than re-use the same request. This makes it easier to understand where to render your components and where to place the Network Boundary.

In practice, this model encourages developers to think about what they want to execute on the server first, before sending the result to the client and making the application interactive.

This concept will become clearer when we look at how you can [interleave client and server components](/docs/app/building-your-application/rendering/composition-patterns) in the same component tree.

---
title: Caching in Next.js
nav_title: Caching
description: An overview of caching mechanisms in Next.js.

Next.js improves your application's performance and reduces costs by caching rendering work and data requests. This page provides an in-depth look at Next.js caching mechanisms, the APIs you can use to configure them, and how they interact with each other.

> **Good to know**: This page helps you understand how Next.js works under the hood but is **not** essential knowledge to be productive with Next.js. Most of Next.js' caching heuristics are determined by your API usage and have defaults for the best performance with zero or minimal configuration.

## Overview

Here's a high-level overview of the different caching mechanisms and their purpose:

Mechanism	What	Where	Purpose	Duration
[Request Memoization](#request-memoization)	Return values of functions	Server	Re-use data in a React Component tree	Per-request lifecycle
[Data Cache](#data-cache)	Data	Server	Store data across user requests and deployments	Persistent (can be revalidated)
[Full Route Cache](#full-route-cache)	HTML and RSC payload	Server	Reduce rendering cost and improve performance	Persistent (can be revalidated)
[Router Cache](#router-cache)	RSC Payload	Client	Reduce server requests on navigation	User session or time-based

By default, Next.js will cache as much as possible to improve performance and reduce cost. This means routes are **statically rendered** and data requests are **cached** unless you opt out. The diagram below shows the default caching behavior: when a route is statically rendered at build time and when a static route is first visited.

<Image
  alt="Diagram showing the default caching behavior in Next.js for the four mechanisms, with HIT, MISS and SET at build time and when a route is first visited."
  srcLight="/docs/light/caching-overview.png"
  srcDark="/docs/dark/caching-overview.png"
  width="1600"
  height="1179"
/>

Caching behavior changes depending on whether the route is statically or dynamically rendered, data is cached or uncached, and whether a request is part of an initial visit or a subsequent navigation. Depending on your use case, you can configure the caching behavior for individual routes and data requests.

## Request Memoization

React extends the [`fetch` API](#fetch) to automatically **memoize** requests that have the same URL and options. This means you can call a fetch function for the same data in multiple places in a React component tree while only executing it once.

<Image
  alt="Deduplicated Fetch Requests"
  srcLight="/docs/light/deduplicated-fetch-requests.png"
  srcDark="/docs/dark/deduplicated-fetch-requests.png"
  width="1600"
  height="857"
/>

For example, if you need to use the same data across a route (e.g. in a Layout, Page, and multiple components), you do not have to fetch data at the top of the tree then forward props between components. Instead, you can fetch data in the components that need it without worrying about the performance implications of making multiple requests across the network for the same data.

```tsx filename="app/example.tsx" switcher
async function getItem() {
  // The `fetch` function is automatically memoized and the result
  // is cached
  const res = await fetch('https://.../item/1')
  return res.json()
}

// This function is called twice, but only executed the first time
const item = await getItem() // cache MISS

// The second call could be anywhere in your route
const item = await getItem() // cache HIT
```

```jsx filename="app/example.js" switcher
async function getItem() {
  // The `fetch` function is automatically memoized and the result
  // is cached
  const res = await fetch('https://.../item/1')
```

  return res.json()
}

// This function is called twice, but only executed the first time
const item = await getItem() // cache MISS

// The second call could be anywhere in your route
const item = await getItem() // cache HIT
```

**How Request Memoization Works**

<Image
  alt="Diagram showing how fetch memoization works during React rendering."
  srcLight="/docs/light/request-memoization.png"
  srcDark="/docs/dark/request-memoization.png"
  width="1600"
  height="742"
/>

- While rendering a route, the first time a particular request is called, its result will not be in memory and it'll be a cache `MISS`.
- Therefore, the function will be executed, and the data will be fetched from the external source, and the result will be stored in memory.
- Subsequent function calls of the request in the same render pass will be a cache `HIT`, and the data will be returned from memory without executing the function.
- Once the route has been rendered and the rendering pass is complete, memory is "reset" and all request memoization entries are cleared.

> **Good to know**:
>
> - Request memoization is a React feature, not a Next.js feature. It's included here to show how it interacts with the other caching mechanisms.
> - Memoization only applies to the `GET` method in `fetch` requests.
> - Memoization only applies to the React Component tree, this means:
>   - It applies to `fetch` requests in `generateMetadata`, `generateStaticParams`, Layouts, Pages, and other Server Components.
>   - It doesn't apply to `fetch` requests in Route Handlers as they are not a part of the React component tree.
> - For cases where `fetch` is not suitable (e.g. some database clients, CMS clients, or GraphQL clients), you can use the [React `cache` function](#react-cache-function) to memoize functions.

### Duration

The cache lasts the lifetime of a server request until the React component tree
```

has finished rendering.

### Revalidating

Since the memoization is not shared across server requests and only applies during rendering, there is no need to revalidate it.

### Opting out

To opt out of memoization in `fetch` requests, you can pass an `AbortController` `signal` to the request.

```js filename="app/example.js"
const { signal } = new AbortController()
fetch(url, { signal })
```

## Data Cache

Next.js has a built-in Data Cache that **persists** the result of data fetches across incoming **server requests** and **deployments**. This is possible because Next.js extends the native `fetch` API to allow each request on the server to set its own persistent caching semantics.

> **Good to know**: In the browser, the `cache` option of `fetch` indicates how a request will interact with the browser's HTTP cache, in Next.js, the `cache` option indicates how a server-side request will interact with the server's Data Cache.

By default, data requests that use `fetch` are **cached**. You can use the [`cache`](#fetch-optionscache) and [`next.revalidate`](#fetch-optionsnextrevalidate) options of `fetch` to configure the caching behavior.

How the Data Cache Works

<Image
  alt="Diagram showing how cached and uncached fetch requests interact with the Data Cache. Cached requests are stored in the Data Cache, and memoized, uncached requests are fetched from the data source, not stored in the Data Cache, and memoized."
  srcLight="/docs/light/data-cache.png"
  srcDark="/docs/dark/data-cache.png"
  width="1600"
  height="661"
/>

- The first time a `fetch` request is called during rendering, Next.js checks the

Data Cache for a cached response.
- If a cached response is found, it's returned immediately and [memoized](#request-memoization).
- If a cached response is not found, the request is made to the data source, the result is stored in the Data Cache, and memoized.
- For uncached data (e.g. `{ cache: 'no-store' }`), the result is always fetched from the data source, and memoized.
- Whether the data is cached or uncached, the requests are always memoized to avoid making duplicate requests for the same data during a React render pass.

> **Differences between the Data Cache and Request Memoization**
>
> While both caching mechanisms help improve performance by re-using cached data, the Data Cache is persistent across incoming requests and deployments, whereas memoization only lasts the lifetime of a request.
>
> With memoization, we reduce the number of **duplicate** requests in the same render pass that have to cross the network boundary from the rendering server to the Data Cache server (e.g. a CDN or Edge Network) or data source (e.g. a database or CMS). With the Data Cache, we reduce the number of requests made to our origin data source.

### Duration

The Data Cache is persistent across incoming requests and deployments unless you revalidate or opt-out.

### Revalidating

Cached data can be revalidated in two ways, with:

- **Time-based Revalidation**: Revalidate data after a certain amount of time has passed and a new request is made. This is useful for data that changes infrequently and freshness is not as critical.
- **On-demand Revalidation:** Revalidate data based on an event (e.g. form submission). On-demand revalidation can use a tag-based or path-based approach to revalidate groups of data at once. This is useful when you want to ensure the latest data is shown as soon as possible (e.g. when content from your headless CMS is updated).

#### Time-based Revalidation

To revalidate data at a timed interval, you can use the `next.revalidate` option of `fetch` to set the cache lifetime of a resource (in seconds).

```js
```

// Revalidate at most every hour
fetch('https://...', { next: { revalidate: *3600* } })
```

Alternatively, you can use [Route Segment Config options](#segment-config-options) to configure all `fetch` requests in a segment or for cases where you're not able to use `fetch`.

**How Time-based Revalidation Works**

<Image
  alt="Diagram showing how time-based revalidation works, after the revalidation period, stale data is returned for the first request, then data is revalidated."
  srcLight="/docs/light/time-based-revalidation.png"
  srcDark="/docs/dark/time-based-revalidation.png"
  width="1600"
  height="1252"
/>

- The first time a fetch request with `revalidate` is called, the data will be fetched from the external data source and stored in the Data Cache.
- Any requests that are called within the specified timeframe (e.g. 60-seconds) will return the cached data.
- After the timeframe, the next request will still return the cached (now stale) data.
  - Next.js will trigger a revalidation of the data in the background.
  - Once the data is fetched successfully, Next.js will update the Data Cache with the fresh data.
  - If the background revalidation fails, the previous data will be kept unaltered.

This is similar to [**stale-while-revalidate**](https://web.dev/stale-while-revalidate/) behavior.

#### On-demand Revalidation

Data can be revalidated on-demand by path ([`revalidatePath`](#revalidatepath)) or by cache tag ([`revalidateTag`](#fetch-optionsnexttags-and-revalidatetag)).

**How On-Demand Revalidation Works**

<Image
  alt="Diagram showing how on-demand revalidation works, the Data Cache is updated with fresh data after a revalidation request."
  srcLight="/docs/light/on-demand-revalidation.png"
  srcDark="/docs/dark/on-demand-revalidation.png"
```

  width="1600"
  height="1082"
/>

- The first time a `fetch` request is called, the data will be fetched from the external data source and stored in the Data Cache.
- When an on-demand revalidation is triggered, the appropriate cache entries will be purged from the cache.
  - This is different from time-based revalidation, which keeps the stale data in the cache until the fresh data is fetched.
- The next time a request is made, it will be a cache `MISS` again, and the data will be fetched from the external data source and stored in the Data Cache.

### Opting out

For individual data fetches, you can opt out of caching by setting the [`cache`](#fetch-optionscache) option to `no-store`. This means data will be fetched whenever `fetch` is called.

```jsx
// Opt out of caching for an individual `fetch` request
fetch(`https://...`, { cache: 'no-store' })
```

Alternatively, you can also use the [Route Segment Config options](#segment-config-options) to opt out of caching for a specific route segment. This will affect all data requests in the route segment, including third-party libraries.

```jsx
// Opt out of caching for all data requests in the route segment
export const dynamic = 'force-dynamic'
```

> **Vercel Data Cache**
>
> If your Next.js application is deployed to Vercel, we recommend reading the [Vercel Data Cache](https://vercel.com/docs/infrastructure/data-cache) documentation for a better understanding of Vercel specific features.

## Full Route Cache

> **Related terms**:
>
> You may see the terms **Automatic Static Optimization**, **Static Site Generation**, or **Static Rendering** being used interchangeably to refer to the process of rendering and caching routes of your application at build time.

Next.js automatically renders and caches routes at build time. This is an optimization that allows you to serve the cached route instead of rendering on the server for every request, resulting in faster page loads.

To understand how the Full Route Cache works, it's helpful to look at how React handles rendering, and how Next.js caches the result:

### 1. React Rendering on the Server

On the server, Next.js uses React's APIs to orchestrate rendering. The rendering work is split into chunks: by individual routes segments and Suspense boundaries.

Each chunk is rendered in two steps:

1. React renders Server Components into a special data format, optimized for streaming, called the **React Server Component Payload**.
2. Next.js uses the React Server Component Payload and Client Component JavaScript instructions to render **HTML** on the server.

This means we don't have to wait for everything to render before caching the work or sending a response. Instead, we can stream a response as work is completed.

> **What is the React Server Component Payload?**
>
> The React Server Component Payload is a compact binary representation of the rendered React Server Components tree. It's used by React on the client to update the browser's DOM. The React Server Component Payload contains:
>
> - The rendered result of Server Components
> - Placeholders for where Client Components should be rendered and references to their JavaScript files
> - Any props passed from a Server Component to a Client Component
>
> To learn more, see the [Server Components](/docs/app/building-your-application/rendering/server-components) documentation.

### 2. Next.js Caching on the Server (Full Route Cache)

<Image
  alt="Default behavior of the Full Route Cache, showing how the React Server Component Payload and HTML are cached on the server for statically rendered routes."
  srcLight="/docs/light/full-route-cache.png"
  srcDark="/docs/dark/full-route-cache.png"

  width="1600"
  height="888"
/>

The default behavior of Next.js is to cache the rendered result (React Server Component Payload and HTML) of a route on the server. This applies to statically rendered routes at build time, or during revalidation.

### 3. React Hydration and Reconciliation on the Client

At request time, on the client:

1. The HTML is used to immediately show a fast non-interactive initial preview of the Client and Server Components.
2. The React Server Components Payload is used to reconcile the Client and rendered Server Component trees, and update the DOM.
3. The JavaScript instructions are used to [hydrate](https://react.dev/reference/react-dom/client/hydrateRoot) Client Components and make the application interactive.

### 4. Next.js Caching on the Client (Router Cache)

The React Server Component Payload is stored in the client-side [Router Cache](#router-cache) - a separate in-memory cache, split by individual route segment. This Router Cache is used to improve the navigation experience by storing previously visited routes and prefetching future routes.

### 5. Subsequent Navigations

On subsequent navigations or during prefetching, Next.js will check if the React Server Components Payload is stored in the Router Cache. If so, it will skip sending a new request to the server.

If the route segments are not in the cache, Next.js will fetch the React Server Components Payload from the server, and populate the Router Cache on the client.

### Static and Dynamic Rendering

Whether a route is cached or not at build time depends on whether it's statically or dynamically rendered. Static routes are cached by default, whereas dynamic routes are rendered at request time, and not cached.

This diagram shows the difference between statically and dynamically rendered routes, with cached and uncached data:

<Image

  alt="How static and dynamic rendering affects the Full Route Cache. Static routes are cached at build time or after data revalidation, whereas dynamic routes are never cached"
  srcLight="/docs/light/static-and-dynamic-routes.png"
  srcDark="/docs/dark/static-and-dynamic-routes.png"
  width="1600"
  height="1314"
/>

Learn more about [static and dynamic rendering](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies).

### Duration

By default, the Full Route Cache is persistent. This means that the render output is cached across user requests.

### Invalidation

There are two ways you can invalidate the Full Route Cache:

- **[Revalidating Data](/docs/app/building-your-application/caching#revalidating)**: Revalidating the [Data Cache](#data-cache), will in turn invalidate the Router Cache by re-rendering components on the server and caching the new render output.
- **Redeploying**: Unlike the Data Cache, which persists across deployments, the Full Route Cache is cleared on new deployments.

### Opting out

You can opt out of the Full Route Cache, or in other words, dynamically render components for every incoming request, by:

- **Using a [Dynamic Function](#dynamic-functions)**: This will opt the route out from the Full Route Cache and dynamically render it at request time. The Data Cache can still be used.
- **Using the `dynamic = 'force-dynamic'` or `revalidate = 0` route segment config options**: This will skip the Full Route Cache and the Data Cache. Meaning components will be rendered and data fetched on every incoming request to the server. The Router Cache will still apply as it's a client-side cache.
- **Opting out of the [Data Cache](#data-cache)**: If a route has a `fetch` request that is not cached, this will opt the route out of the Full Route Cache. The data for the specific `fetch` request will be fetched for every incoming request. Other `fetch` requests that do not opt out of caching will still be cached in the Data Cache. This allows for a hybrid of cached and uncached data.

## Router Cache

> **Related Terms:**
>
> You may see the Router Cache being referred to as **Client-side Cache** or **Prefetch Cache**. While **Prefetch Cache** refers to the prefetched route segments, **Client-side Cache** refers to the whole Router cache, which includes both visited and prefetched segments.
> This cache specifically applies to Next.js and Server Components, and is different to the browser's [bfcache](https://web.dev/bfcache/), though it has a similar result.

Next.js has an in-memory client-side cache that stores the React Server Component Payload, split by individual route segments, for the duration of a user session. This is called the Router Cache.

How the Router Cache Works

<Image
  alt="How the Router cache works for static and dynamic routes, showing MISS and HIT for initial and subsequent navigations."
  srcLight="/docs/light/router-cache.png"
  srcDark="/docs/dark/router-cache.png"
  width="1600"
  height="1375"
/>

As a user navigates between routes, Next.js caches visited route segments and [prefetches](/docs/app/building-your-application/routing/linking-and-navigating#1-prefetching) the routes the user is likely to navigate to (based on `<Link>` components in their viewport).

This results in an improved navigation experience for the user:

- Instant backward/forward navigation because visited routes are cached and fast navigation to new routes because of prefetching and [partial rendering](/docs/app/building-your-application/routing/linking-and-navigating#3-partial-rendering).
- No full-page reload between navigations, and React state and browser state are preserved.

> **Difference between the Router Cache and Full Route Cache**:
>
> The Router Cache temporarily stores the React Server Component Payload in the browser for the duration of a user session, whereas the Full Route Cache persistently stores the React Server Component Payload and HTML on the

server across multiple user requests.
>
> While the Full Route Cache only caches statically rendered routes, the Router Cache applies to both statically and dynamically rendered routes.

### Duration

The cache is stored in the browser's temporary memory. Two factors determine how long the router cache lasts:

- **Session**: The cache persists across navigation. However, it's cleared on page refresh.
- **Automatic Invalidation Period**: The cache of an individual segment is automatically invalidated after a specific time. The duration depends on whether the route is [statically](/docs/app/building-your-application/rendering/server-components#static-rendering-default) or [dynamically](/docs/app/building-your-application/rendering/server-components#dynamic-rendering) rendered:
  - **Dynamically Rendered**: 30 seconds
  - **Statically Rendered**: 5 minutes

While a page refresh will clear **all** cached segments, the automatic invalidation period only affects the individual segment from the time it was last accessed or created.

By adding `prefetch={*true*}` or calling `router.prefetch` for a dynamically rendered route, you can opt into caching for 5 minutes.

### Invalidation

There are two ways you can invalidate the Router Cache:

- In a **Server Action**:
  - Revalidating data on-demand by path with ([`revalidatePath`](/docs/app/api-reference/functions/revalidatePath)) or by cache tag with ([`revalidateTag`](/docs/app/api-reference/functions/revalidateTag))
  - Using [`cookies.set`](/docs/app/api-reference/functions/cookies#cookiessetname-value-options) or [`cookies.delete`](/docs/app/api-reference/functions/cookies#deleting-cookies) invalidates the Router Cache to prevent routes that use cookies from becoming stale (e.g. authentication).
- Calling [`router.refresh`](/docs/app/api-reference/functions/use-router) will invalidate the Router Cache and make a new request to the server for the current route.

### Opting out

It's not possible to opt out of the Router Cache.

You can opt out of **prefetching** by setting the `prefetch` prop of the `<Link>` component to `false`. However, this will still temporarily store the route segments for 30s to allow instant navigation between nested segments, such as tab bars, or back and forward navigation. Visited routes will still be cached.

## Cache Interactions

When configuring the different caching mechanisms, it's important to understand how they interact with each other:

### Data Cache and Full Route Cache

- Revalidating or opting out of the Data Cache **will** invalidate the Full Route Cache, as the render output depends on data.
- Invalidating or opting out of the Full Route Cache **does not** affect the Data Cache. You can dynamically render a route that has both cached and uncached data. This is useful when most of your page uses cached data, but you have a few components that rely on data that needs to be fetched at request time. You can dynamically render without worrying about the performance impact of re-fetching all the data.

### Data Cache and Client-side Router cache

- Revalidating the Data Cache in a [Route Handler](/docs/app/building-your-application/routing/route-handlers) **will not** immediately invalidate the Router Cache as the Route Handler isn't tied to a specific route. This means Router Cache will continue to serve the previous payload until a hard refresh, or the automatic invalidation period has elapsed.
- To immediately invalidate the Data Cache and Router cache, you can use [`revalidatePath`](#revalidatepath) or [`revalidateTag`](#fetch-optionsnexttags-and-revalidatetag) in a [Server Action](/docs/app/building-your-application/data-fetching/forms-and-mutations).

## APIs

The following table provides an overview of how different Next.js APIs affect caching:

API	Router Cache	Full Route Cache	Data Cache	React Cache
[`<Link prefetch>`](#link)	Cache			

[`router.prefetch`](#routerprefetch)	Cache			
[`router.refresh`](#routerrefresh)	Revalidate			
[`fetch`](#fetch)			Cache	Cache
[`fetch` `options.cache`](#fetch-optionscache)			Cache or Opt out	
[`fetch` `options.next.revalidate`](#fetch-optionsnextrevalidate)		Revalidate	Revalidate	
[`fetch` `options.next.tags`](#fetch-optionsnexttags-and-revalidatetag)		Cache	Cache	
[`revalidateTag`](#fetch-optionsnexttags-and-revalidatetag)	Revalidate (Server Action)	Revalidate	Revalidate	
[`revalidatePath`](#revalidatepath)	Revalidate (Server Action)	Revalidate	Revalidate	
[`const revalidate`](#segment-config-options)		Revalidate or Opt out	Revalidate or Opt out	
[`const dynamic`](#segment-config-options)		Cache or Opt out	Cache or Opt out	
[`cookies`](#cookies)	Revalidate (Server Action)	Opt out		
[`headers`, `useSearchParams`, `searchParams`](#dynamic-functions)		Opt out		
[`generateStaticParams`](#generatestaticparams)		Cache		
[`React.cache`](#react-cache-function)				Cache
[`unstable_cache`](/docs/app/api-reference/functions/unstable_cache)				

### `<Link>`

By default, the `<Link>` component automatically prefetches routes from the Full Route Cache and adds the React Server Component Payload to the Router Cache.

To disable prefetching, you can set the `prefetch` prop to `false`. But this will not skip the cache permanently, the route segment will still be cached client-side when the user visits the route.

Learn more about the [`<Link>` component](/docs/app/api-reference/components/link).

### `router.prefetch`

The `prefetch` option of the `useRouter` hook can be used to manually prefetch a route. This adds the React Server Component Payload to the Router Cache.

See the [`useRouter` hook](/docs/app/api-reference/functions/use-router) API reference.

### `router.refresh`

The `refresh` option of the `useRouter` hook can be used to manually refresh a route. This completely clears the Router Cache, and makes a new request to the server for the current route. `refresh` does not affect the Data or Full Route Cache.

The rendered result will be reconciled on the client while preserving React state and browser state.

See the [`useRouter` hook](/docs/app/api-reference/functions/use-router) API reference.

### `fetch`

Data returned from `fetch` is automatically cached in the Data Cache.

```jsx
// Cached by default. `force-cache` is the default option and can be ommitted.
fetch(`https://...`, { cache: 'force-cache' })
```

See the [`fetch` API Reference](/docs/app/api-reference/functions/fetch) for more options.

### `fetch options.cache`

You can opt out individual `fetch` requests of data caching by setting the `cache` option to `no-store`:

```jsx
// Opt out of caching
fetch(`https://...`, { cache: 'no-store' })
```

Since the render output depends on data, using `cache: 'no-store'` will also skip the Full Route Cache for the route where the `fetch` request is used. That is, the route will be dynamically rendered every request, but you can still have other cached data requests in the same route.

See the [`fetch` API Reference](/docs/app/api-reference/functions/fetch) for more options.

### `fetch options.next.revalidate`

You can use the `next.revalidate` option of `fetch` to set the revalidation period (in seconds) of an individual `fetch` request. This will revalidate the Data Cache, which in turn will revalidate the Full Route Cache. Fresh data will be fetched, and components will be re-rendered on the server.

```jsx
// Revalidate at most after 1 hour
fetch(`https://...`, { next: { revalidate: 3600 } })
```

See the [`fetch` API reference](/docs/app/api-reference/functions/fetch) for more options.

### `fetch options.next.tags` and `revalidateTag`

Next.js has a cache tagging system for fine-grained data caching and revalidation.

1. When using `fetch` or [`unstable_cache`](/docs/app/api-reference/functions/unstable_cache), you have the option to tag cache entries with one or more tags.
2. Then, you can call `revalidateTag` to purge the cache entries associated with that tag.

For example, you can set a tag when fetching data:

```jsx
// Cache data with a tag
fetch(`https://...`, { next: { tags: ['a', 'b', 'c'] } })
```

Then, call `revalidateTag` with a tag to purge the cache entry:

```jsx
// Revalidate entries with a specific tag
revalidateTag('a')
```

There are two places you can use `revalidateTag`, depending on what you're trying to achieve:

1. [Route Handlers](/docs/app/building-your-application/routing/route-handlers) - to revalidate data in response of a third party event (e.g. webhook).

This will not invalidate the Router Cache immediately as the Router Handler isn't tied to a specific route.
2. [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations) - to revalidate data after a user action (e.g. form submission). This will invalidate the Router Cache for the associated route.

### `revalidatePath`

`revalidatePath` allows you manually revalidate data **and** re-render the route segments below a specific path in a single operation. Calling the `revalidatePath` method revalidates the Data Cache, which in turn invalidates the Full Route Cache.

```jsx
revalidatePath('/')
```

There are two places you can use `revalidatePath`, depending on what you're trying to achieve:

1. [Route Handlers](/docs/app/building-your-application/routing/route-handlers) - to revalidate data in response to a third party event (e.g. webhook).
2. [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations) - to revalidate data after a user interaction (e.g. form submission, clicking a button).

See the [`revalidatePath` API reference](/docs/app/api-reference/functions/revalidatePath) for more information.

> **`revalidatePath`** vs. **`router.refresh`**:
>
> Calling `router.refresh` will clear the Router cache, and re-render route segments on the server without invalidating the Data Cache or the Full Route Cache.
>
> The difference is that `revalidatePath` purges the Data Cache and Full Route Cache, whereas `router.refresh()` does not change the Data Cache and Full Route Cache, as it is a client-side API.

### Dynamic Functions

`cookies`, `headers`, `useSearchParams`, and `searchParams` are all dynamic functions that depend on runtime incoming request information. Using them will opt a route out of the Full Route Cache, in other words, the route will be dynamically rendered.

#### `cookies`

Using `cookies.set` or `cookies.delete` in a Server Action invalidates the Router Cache to prevent routes that use cookies from becoming stale (e.g. to reflect authentication changes).

See the [`cookies`](/docs/app/api-reference/functions/cookies) API reference.

### Segment Config Options

The Route Segment Config options can be used to override the route segment defaults or when you're not able to use the `fetch` API (e.g. database client or 3rd party libraries).

The following Route Segment Config options will opt out of the Data Cache and Full Route Cache:

- `const dynamic = 'force-dynamic'`
- `const revalidate = 0`

See the [Route Segment Config](/docs/app/api-reference/file-conventions/route-segment-config) documentation for more options.

### `generateStaticParams`

For [dynamic segments](/docs/app/building-your-application/routing/dynamic-routes) (e.g. `app/blog/[slug]/page.js`), paths provided by `generateStaticParams` are cached in the Full Route Cache at build time. At request time, Next.js will also cache paths that weren't known at build time the first time they're visited.

You can disable caching at request time by using `export const dynamicParams = false` option in a route segment. When this config option is used, only paths provided by `generateStaticParams` will be served, and other routes will 404 or match (in the case of [catch-all routes](/docs/app/building-your-application/routing/dynamic-routes#catch-all-segments)).

See the [`generateStaticParams` API reference](/docs/app/api-reference/functions/generate-static-params).

### React `cache` function

The React `cache` function allows you to memoize the return value of a function, allowing you to call the same function multiple times while only executing it once.

Since `fetch` requests are automatically memoized, you do not need to wrap it in React `cache`. However, you can use `cache` to manually memoize data requests for use cases when the `fetch` API is not suitable. For example, some database clients, CMS clients, or GraphQL clients.

```tsx filename="utils/get-item.ts" switcher
import { cache } from 'react'
import db from '@/lib/db'

export const getItem = cache(async (id: string) => {
  const item = await db.item.findUnique({ id })
  return item
})
```

```jsx filename="utils/get-item.js" switcher
import { cache } from 'react'
import db from '@/lib/db'

export const getItem = cache(async (id) => {
  const item = await db.item.findUnique({ id })
  return item
})
```

---
title: CSS Modules
description: Style your Next.js Application with CSS Modules.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<PagesOnly>

<details open>
  <summary>Examples</summary>

- [Basic CSS Example](https://github.com/vercel/next.js/tree/canary/examples/basic-css)

</details>

</PagesOnly>

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

<AppOnly>
CSS Modules can be imported into any file inside the `app` directory:

```tsx filename="app/dashboard/layout.tsx" switcher
import styles from './styles.module.css'

export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return <section className={styles.dashboard}>{children}</section>
}
```

```jsx filename="app/dashboard/layout.js" switcher
import styles from './styles.module.css'

export default function DashboardLayout({ children }) {
  return <section className={styles.dashboard}>{children}</section>
}
```

```css filename="app/dashboard/styles.module.css"
.dashboard {
  padding: 24px;
}
```

</AppOnly>

<PagesOnly>

For example, consider a reusable `Button` component in the `components/` folder:

First, create `components/Button.module.css` with the following content:

```css filename="Button.module.css"
```

```css
/*
You do not need to worry about .error {} colliding with any other `.css` or
`.module.css` files!
*/
.error {
  color: white;
  background-color: red;
}
```

Then, create `components/Button.js`, importing and using the above CSS file:

```jsx filename="components/Button.js"
import styles from './Button.module.css'

export function Button() {
  return (
    <button
      type="button"
      // Note how the "error" class is accessed as a property on the imported
      // `styles` object.
      className={styles.error}
    >
      Destroy
    </button>
  )
}
```

</PagesOnly>

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

<AppOnly>
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`:

```css
body {
  padding: 20px 20px 60px;
  max-width: 680px;
  margin: 0 auto;
}
```

Inside the root layout (`app/layout.js`), import the `global.css` stylesheet to apply the styles to every route in your application:

```tsx filename="app/layout.tsx" switcher
// These styles apply to every route in the application
import './global.css'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
// These styles apply to every route in the application
import './global.css'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

</AppOnly>

<PagesOnly>

To add a stylesheet to your application, import the CSS file within `pages/_app.js`.

For example, consider the following stylesheet named `styles.css`:

```css filename="styles.css"
body {
  font-family: 'SF Pro Text', 'SF Pro Icons', 'Helvetica Neue', 'Helvetica',
    'Arial', sans-serif;
  padding: 20px 20px 60px;
  max-width: 680px;
  margin: 0 auto;
}
```

Create a [`pages/_app.js` file](/docs/pages/building-your-application/routing/custom-app) if not already present.
Then, [`import`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/import) the `styles.css` file.

```jsx filename="pages/_app.js"
import '../styles.css'

// This default export is required in a new `pages/_app.js` file.
export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

These styles (`styles.css`) will apply to all pages and components in your application.
Due to the global nature of stylesheets, and to avoid conflicts, you may **only import them inside [`pages/_app.js`](/docs/pages/building-your-application/routing/custom-app)**.

In development, expressing stylesheets this way allows your styles to be hot reloaded as you edit them—meaning you can keep application state.

In production, all CSS files will be automatically concatenated into a single minified `.css` file. The order that the CSS is concatenated will match the order the CSS is imported into the `_app.js` file. Pay special attention to imported JS modules that include their own CSS; the JS module's CSS will be concatenated following the same ordering rules as imported CSS files. For example:

```jsx
import '../styles.css'
// The CSS in ErrorBoundary depends on the global CSS in styles.css,
// so we import it after styles.css.
import ErrorBoundary from '../components/ErrorBoundary'

export default function MyApp({ Component, pageProps }) {
  return (
    <ErrorBoundary>
      <Component {...pageProps} />
    </ErrorBoundary>
  )
}
```

</PagesOnly>

## External Stylesheets

<AppOnly>

Stylesheets published by external packages can be imported anywhere in the `app` directory, including colocated components:

```tsx filename="app/layout.tsx" switcher
import 'bootstrap/dist/css/bootstrap.css'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body className="container">{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import 'bootstrap/dist/css/bootstrap.css'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
```

    <body className="container">{children}</body>
  </html>
)
}

> **Good to know**: External stylesheets must be directly imported from an npm package or downloaded and colocated with your codebase. You cannot use `<link rel="stylesheet" />`.

</AppOnly>

<PagesOnly>

Next.js allows you to import CSS files from a JavaScript file.
This is possible because Next.js extends the concept of [`import`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/import) beyond JavaScript.

### Import styles from `node_modules`

Since Next.js **9.5.4**, importing a CSS file from `node_modules` is permitted anywhere in your application.

For global stylesheets, like `bootstrap` or `nprogress`, you should import the file inside `pages/_app.js`.
For example:

```jsx filename="pages/_app.js"
import 'bootstrap/dist/css/bootstrap.css'

export default function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

For importing CSS required by a third-party component, you can do so in your component. For example:

```jsx filename="components/example-dialog.js"
import { useState } from 'react'
import { Dialog } from '@reach/dialog'
import VisuallyHidden from '@reach/visually-hidden'
import '@reach/dialog/styles.css'

function ExampleDialog(props) {
  const [showDialog, setShowDialog] = useState(false)
  const open = () => setShowDialog(true)
  const close = () => setShowDialog(false)

  return (
    <div>
      <button onClick={open}>Open Dialog</button>
      <Dialog isOpen={showDialog} onDismiss={close}>
        <button className="close-button" onClick={close}>
          <VisuallyHidden>Close</VisuallyHidden>
          <span aria-hidden>×</span>
        </button>
        <p>Hello there. I am a dialog</p>
      </Dialog>
    </div>
  )
}
```

</PagesOnly>

## 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](/docs/architecture/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](/docs/architecture/fast-refresh).

---
title: Tailwind CSS
description: Style your Next.js Application using Tailwind CSS.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<PagesOnly>

<details open>
  <summary>Examples</summary>

- [With Tailwind CSS](https://github.com/vercel/next.js/tree/canary/examples/with-tailwindcss)

</details>

</PagesOnly>

[Tailwind CSS](https://tailwindcss.com/) 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:

```bash filename="Terminal"
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

## Configuring Tailwind

Inside `tailwind.config.js`, add paths to the files that will use Tailwind CSS class names:

```js filename="tailwind.config.js"
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    './app/**/*.{js,ts,jsx,tsx,mdx}', // Note the addition of the `app` directory.
    './pages/**/*.{js,ts,jsx,tsx,mdx}',
    './components/**/*.{js,ts,jsx,tsx,mdx}',

    // Or if using `src` directory:
    './src/**/*.{js,ts,jsx,tsx,mdx}',
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

You do not need to modify `postcss.config.js`.

<AppOnly>

## Importing Styles

Add the [Tailwind CSS directives](https://tailwindcss.com/docs/functions-and-directives#directives) that Tailwind will use to inject its generated styles to a [Global Stylesheet](/docs/app/building-your-application/styling/css-modules#global-styles) in your application, for example:

```css filename="app/globals.css"
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Inside the [root layout](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required) (`app/layout.tsx`), import the `globals.css` stylesheet to apply the styles to every route in your application.

```tsx filename="app/layout.tsx" switcher
import type { Metadata } from 'next'

// These styles apply to every route in the application
import './globals.css'

export const metadata: Metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
```

// These styles apply to every route in the application
import './globals.css'

export const metadata = {
  title: 'Create Next App',
  description: 'Generated by create next app',
}

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
)
}

## Using Classes

After installing Tailwind CSS and adding the global styles, you can use Tailwind's utility classes in your application.

```tsx filename="app/page.tsx" switcher
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

```jsx filename="app/page.js" switcher
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

</AppOnly>

<PagesOnly>

## Importing Styles

Add the [Tailwind CSS directives](https://tailwindcss.com/docs/functions-and-directives#directives) that Tailwind will use to inject its generated styles to a [Global Stylesheet](/docs/pages/building-your-application/styling/css-modules#global-styles) in your application, for example:

```css filename="styles/globals.css"
@tailwind base;
```

@tailwind components;
@tailwind utilities;

Inside the [custom app file](/docs/pages/building-your-application/routing/custom-app) (`pages/_app.js`), import the `globals.css` stylesheet to apply the styles to every route in your application.

```tsx filename="pages/_app.tsx" switcher
// These styles apply to every route in the application
import '@/styles/globals.css'
import type { AppProps } from 'next/app'

export default function App({ Component, pageProps }: AppProps) {
  return <Component {...pageProps} />
}
```

```jsx filename="pages/_app.js" switcher
// These styles apply to every route in the application
import '@/styles/globals.css'

export default function App({ Component, pageProps }) {
  return <Component {...pageProps} />
}
```

## Using Classes

After installing Tailwind CSS and adding the global styles, you can use Tailwind's utility classes in your application.

```tsx filename="pages/index.tsx" switcher
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

```jsx filename="pages/index.js" switcher
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

</PagesOnly>

## Usage with Turbopack

As of Next.js 13.1, Tailwind CSS and PostCSS are supported with [Turbopack](https://turbo.build/pack/docs/features/css#tailwind-css).

---
title: CSS-in-JS
description: Use CSS-in-JS libraries with Next.js

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<AppOnly>

> **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](https://react.dev/blog/2022/03/29/react-v18#what-is-concurrent-react).
>
> 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 (alphabetical):

- [`kuma-ui`](https://kuma-ui.com)
- [`@mui/material`](https://mui.com/material-ui/guides/next-js-app-router/)
- [`pandacss`](https://panda-css.com)
- [`styled-jsx`](#styled-jsx)
- [`styled-components`](#styled-components)
- [`style9`](https://github.com/johanholmerin/style9)
- [`tamagui`](https://tamagui.dev/docs/guides/next-js#server-components)
- [`tss-react`](https://tss-react.dev/)
- [`vanilla-extract`](https://github.com/vercel/next.js/tree/canary/examples/with-vanilla-extract)

The following are currently working on support:

- [`emotion`](https://github.com/emotion-js/emotion/issues/2928)

> **Good to know**: 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](/docs/app/building-your-application/styling/css-modules) or other solutions that output CSS files, like PostCSS or [Tailwind CSS](/docs/app/building-your-application/styling/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:

```tsx filename="app/registry.tsx" switcher
'use client'

import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/navigation'
import { StyleRegistry, createStyleRegistry } from 'styled-jsx'

export default function StyledJsxRegistry({
  children,
}: {
  children: React.ReactNode
}) {
  // Only create stylesheet once with lazy initial state
  // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
  const [jsxStyleRegistry] = useState(() => createStyleRegistry())

  useServerInsertedHTML(() => {
    const styles = jsxStyleRegistry.styles()
    jsxStyleRegistry.flush()
    return <>{styles}</>
  })

  return <StyleRegistry registry={jsxStyleRegistry}>{children}</StyleRegistry>
}
```

```jsx filename="app/registry.js" switcher
'use client'

import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/navigation'
import { StyleRegistry, createStyleRegistry } from 'styled-jsx'

export default function StyledJsxRegistry({ children }) {
  // Only create stylesheet once with lazy initial state
  // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
  const [jsxStyleRegistry] = useState(() => createStyleRegistry())

  useServerInsertedHTML(() => {
    const styles = jsxStyleRegistry.styles()
    jsxStyleRegistry.flush()
    return <>{styles}</>
  })

  return <StyleRegistry registry={jsxStyleRegistry}>{children}</StyleRegistry>
}
```

Then, wrap your [root layout](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required) with the registry:

```tsx filename="app/layout.tsx" switcher
import StyledJsxRegistry from './registry'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <StyledJsxRegistry>{children}</StyledJsxRegistry>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import StyledJsxRegistry from './registry'
```

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <StyledJsxRegistry>{children}</StyledJsxRegistry>
      </body>
    </html>
)
}
~~~

[View an example here](https://github.com/vercel/app-playground/tree/main/app/styling/styled-jsx).

Styled Components

Below is an example of how to configure `styled-components@6` or newer:

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.

```tsx filename="lib/registry.tsx" switcher
'use client'

import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/navigation'
import { ServerStyleSheet, StyleSheetManager } from 'styled-components'

export default function StyledComponentsRegistry({
  children,
}: {
  children: React.ReactNode
}) {
  // Only create stylesheet once with lazy initial state
  // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
  const [styledComponentsStyleSheet] = useState(() => new
ServerStyleSheet())

  useServerInsertedHTML(() => {
    const styles = styledComponentsStyleSheet.getStyleElement()
    styledComponentsStyleSheet.instance.clearTag()
    return <>{styles}</>
  })
~~~

```
  if (typeof window !== 'undefined') return <>{children}</>

  return (
    <StyleSheetManager sheet={styledComponentsStyleSheet.instance}>
      {children}
    </StyleSheetManager>
  )
}
```

```jsx filename="lib/registry.js" switcher
'use client'

import React, { useState } from 'react'
import { useServerInsertedHTML } from 'next/navigation'
import { ServerStyleSheet, StyleSheetManager } from 'styled-components'

export default function StyledComponentsRegistry({ children }) {
  // Only create stylesheet once with lazy initial state
  // x-ref: https://reactjs.org/docs/hooks-reference.html#lazy-initial-state
  const [styledComponentsStyleSheet] = useState(() => new
ServerStyleSheet())

  useServerInsertedHTML(() => {
    const styles = styledComponentsStyleSheet.getStyleElement()
    styledComponentsStyleSheet.instance.clearTag()
    return <>{styles}</>
  })

  if (typeof window !== 'undefined') return <>{children}</>

  return (
    <StyleSheetManager sheet={styledComponentsStyleSheet.instance}>
      {children}
    </StyleSheetManager>
  )
}
```

Wrap the `children` of the root layout with the style registry component:

```tsx filename="app/layout.tsx" switcher
import StyledComponentsRegistry from './lib/registry'

export default function RootLayout({
  children,
```

}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <StyledComponentsRegistry>{children}</StyledComponentsRegistry>
      </body>
    </html>
)
}

```jsx filename="app/layout.js" switcher
import StyledComponentsRegistry from './lib/registry'

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <StyledComponentsRegistry>{children}</StyledComponentsRegistry>
      </body>
    </html>
  )
}
```

[View an example here](https://github.com/vercel/app-playground/tree/main/app/styling/styled-components).

> **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.

</AppOnly>

<PagesOnly>

<details>
  <summary>Examples</summary>

- [Styled JSX](https://github.com/vercel/next.js/tree/canary/examples/with-styled-jsx)
- [Styled Components](https://github.com/vercel/next.js/tree/canary/examples/with-styled-components)
- [Emotion](https://github.com/vercel/next.js/tree/canary/examples/with-emotion)
- [Linaria](https://github.com/vercel/next.js/tree/canary/examples/with-linaria)
- [Tailwind CSS + Emotion](https://github.com/vercel/next.js/tree/canary/examples/with-tailwindcss-emotion)
- [Styletron](https://github.com/vercel/next.js/tree/canary/examples/with-styletron)
- [Cxs](https://github.com/vercel/next.js/tree/canary/examples/with-cxs)
- [Aphrodite](https://github.com/vercel/next.js/tree/canary/examples/with-aphrodite)
- [Fela](https://github.com/vercel/next.js/tree/canary/examples/with-fela)
- [Stitches](https://github.com/vercel/next.js/tree/canary/examples/with-stitches)

</details>

It's possible to use any existing CSS-in-JS solution.The simplest one is inline styles:

```jsx
function HiThere() {
  return <p style={{ color: 'red' }}>hi there</p>
}

export default HiThere
```

We bundle [styled-jsx](https://github.com/vercel/styled-jsx) to provide support for isolated scoped CSS.
The aim is to support "shadow CSS" similar to Web Components, which unfortunately [do not support server-rendering and are JS-only](https://github.com/w3c/webcomponents/issues/71).

See the above examples for other popular CSS-in-JS solutions (like Styled Components).

A component using `styled-jsx` looks like this:

```jsx
```

```
function HelloWorld() {
  return (
    <div>
      Hello world
      <p>scoped!</p>
      <style jsx>{`
        p {
          color: blue;
        }
        div {
          background: red;
        }
        @media (max-width: 600px) {
          div {
            background: blue;
          }
        }
      `}</style>
      <style global jsx>{`
        body {
          background: black;
        }
      `}</style>
    </div>
  )
}

export default HelloWorld
```

Please see the [styled-jsx documentation](https://github.com/vercel/styled-jsx) for more examples.

### Disabling JavaScript

Yes, if you disable JavaScript the CSS will still be loaded in the production build (`next start`). During development, we require JavaScript to be enabled to provide the best developer experience with [Fast Refresh](https://nextjs.org/blog/next-9-4#fast-refresh).

</PagesOnly>

---
title: Sass
description: Style your Next.js application using Sass.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js has built-in support for integrating with Sass after the package is installed 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`](https://github.com/sass/sass):

```bash filename="Terminal"
npm install --save-dev sass
```

> **Good to know**:
>
> Sass supports [two different syntaxes](https://sass-lang.com/documentation/syntax), each with their own extension.
> The `.scss` extension requires you use the [SCSS syntax](https://sass-lang.com/documentation/syntax#scss),
> while the `.sass` extension requires you use the [Indented Syntax ("Sass")](https://sass-lang.com/documentation/syntax#the-indented-syntax).
>
> 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 filename="next.config.js"
const path = require('path')

module.exports = {
  sassOptions: {
    includePaths: [path.join(__dirname, 'styles')],
  },
}
```

### Sass Variables

Next.js supports Sass variables exported from CSS Module files.

For example, using the exported `primaryColor` Sass variable:

```scss filename="app/variables.module.scss"
$primary-color: #64ff00;

:export {
  primaryColor: $primary-color;
}
```

<AppOnly>

```jsx filename="app/page.js"
// maps to root `/` URL

import variables from './variables.module.scss'

export default function Page() {
  return <h1 style={{ color: variables.primaryColor }}>Hello, Next.js!</h1>
}
```

</AppOnly>

<PagesOnly>

```jsx filename="pages/_app.js"
import variables from '../styles/variables.module.scss'

export default function MyApp({ Component, pageProps }) {
  return (
    <Layout color={variables.primaryColor}>
      <Component {...pageProps} />
    </Layout>
  )
}
```

</PagesOnly>

---
title: Styling
description: Learn the different ways you can style your Next.js application.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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:

```
---
title: Image Optimization
nav_title: Images
description: Optimize your images with the built-in `next/image` component.
related:
  title: API Reference
  description: Learn more about the next/image API.
  links:
    - app/api-reference/components/image
---
```

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<details>
  <summary>Examples</summary>

- [Image Component](https://github.com/vercel/next.js/tree/canary/examples/image-component)

</details>

According to [Web Almanac](https://almanac.httparchive.org), images account for a huge portion of the typical website’s [page weight](https://almanac.httparchive.org/en/2022/page-weight#content-type-and-file-formats) and can have a sizable impact on your website's [LCP performance](https://almanac.httparchive.org/en/2022/performance#lcp-image-optimization).

The Next.js Image component extends the HTML `<img>` element with features for automatic image optimization:

- **Size Optimization:** Automatically serve correctly sized images for each device, using modern image formats like WebP and AVIF.
- **Visual Stability:** Prevent [layout shift](/learn/seo/web-performance/cls) 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

> **🎥 Watch:** Learn more about how to use `next/image` → [YouTube (9 minutes)](https://youtu.be/IU_qq_c_lKA).

## Usage

```js
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](#image-sizing) the `width` and `height` of your image based on the imported file. These values are used to prevent [Cumulative Layout Shift](https://nextjs.org/learn/seo/web-performance/cls) while your image is loading.

<AppOnly>

```jsx filename="app/page.js"
import Image from 'next/image'
import profilePic from './me.png'

export default function Page() {
  return (
    <Image
      src={profilePic}
```

      alt="Picture of the author"
      // width={500} automatically provided
      // height={500} automatically provided
      // blurDataURL="data:..." automatically provided
      // placeholder="blur" // Optional blur-up while loading
    />
)
}

</AppOnly>

<PagesOnly>

```jsx filename="pages/index.js"
import Image from 'next/image'
import profilePic from '../public/me.png'

export default function Page() {
  return (
    <Image
      src={profilePic}
      alt="Picture of the author"
      // width={500} automatically provided
      // height={500} automatically provided
      // blurDataURL="data:..." automatically provided
      // placeholder="blur" // Optional blur-up while loading
    />
  )
}
```

</PagesOnly>

> **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`](/docs/app/api-reference/components/image#width), [`height`](/docs/app/api-reference/components/image#height) and optional [`blurDataURL`](/docs/app/api-reference/components/image#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](#image-sizing).

```jsx filename="app/page.js"
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="https://s3.amazonaws.com/my-bucket/profile.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

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:

```js filename="next.config.js"
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 's3.amazonaws.com',
        port: '',
        pathname: '/my-bucket/**',
      },
    ],
  },
}
```

Learn more about [`remotePatterns`](/docs/app/api-reference/components/image#remotepatterns) configuration. If you want to use relative URLs for the image `src`, use a [`loader`](/docs/app/api-reference/components/image#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`](/docs/app/api-reference/components/image#remotepatterns) configuration.

### Loaders

Note that in the [example earlier](#local-images), a partial URL (`"/me.png"`) is provided for a local 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](https://developer.mozilla.org/docs/Web/API/HTMLImageElement/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](/docs/app/api-reference/components/image#loader), or at the application level with the [`loaderFile` configuration](/docs/app/api-reference/components/image#loaderfile).

## Priority

You should add the `priority` property to the image that will be the [Largest Contentful Paint (LCP) element](https://web.dev/lcp/#what-elements-are-considered) 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:

<PagesOnly>

```jsx filename="app/page.js"
import Image from 'next/image'

export default function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src="/me.png"
        alt="Picture of the author"
        width={500}
        height={500}
        priority
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}
```

</PagesOnly>

<AppOnly>

```jsx filename="app/page.js"
import Image from 'next/image'
import profilePic from '../public/me.png'

export default function Page() {
  return <Image src={profilePic} alt="Picture of the author" priority />
}
```

</AppOnly>

See more about priority in the [`next/image` component documentation](/docs/app/api-reference/components/image#priority).

## 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](https://web.dev/cls/). The way to avoid image-based layout shifts is to [always size your images](https://web.dev/optimize-cls/#images-without-dimensions). 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](#local-images)
2. Explicitly, by including a [`width`](/docs/app/api-reference/components/image#width) and [`height`](/docs/app/api-reference/components/image#height) property
3. Implicitly, by using [fill](/docs/app/api-reference/components/image#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`](/docs/app/api-reference/components/image#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`](/docs/app/api-reference/components/image#sizes) prop to match any media query break points. You can also use [`object-fit`](https://developer.mozilla.org/docs/Web/CSS/object-fit) with `fill`, `contain`, or `cover`, and [`object-position`](https://developer.mozilla.org/docs/Web/CSS/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 `<img>` elements.

## Styling

Styling the Image component is similar to styling a normal `<img>` 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](/docs/app/building-your-application/styling/css-modules), a [global stylesheet](/docs/app/building-your-application/styling/css-modules#global-styles), etc.
  - You can also use the `style` prop to assign inline styles.
  - You cannot use [styled-jsx](/docs/app/building-your-application/styling/css-in-js) 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.

## Examples

### Responsive

<Image
  alt="Responsive image filling the width and height of its parent container"
  srcLight="/docs/light/responsive-image.png"
  srcDark="/docs/dark/responsive-image.png"
  width="1600"
  height="629"
/>

```jsx
import Image from 'next/image'
import mountains from '../public/mountains.jpg'

export default function Responsive() {
  return (
    <div style={{ display: 'flex', flexDirection: 'column' }}>
      <Image
        alt="Mountains"
        // Importing an image will
        // automatically set the width and height
        src={mountains}
        sizes="100vw"
        // Make the image display full width
        style={{
          width: '100%',
          height: 'auto',
        }}
      />
    </div>
  )
}
```

### Fill Container

```
<Image
  alt="Grid of images filling parent container width"
  srcLight="/docs/light/fill-container.png"
  srcDark="/docs/dark/fill-container.png"
  width="1600"
  height="529"
/>
```

```jsx
import Image from 'next/image'
import mountains from '../public/mountains.jpg'

export default function Fill() {
  return (
    <div
      style={{
        display: 'grid',
        gridGap: '8px',
        gridTemplateColumns: 'repeat(auto-fit, minmax(400px, auto))',
      }}
    >
      <div style={{ position: 'relative', height: '400px' }}>
        <Image
          alt="Mountains"
          src={mountains}
          fill
          sizes="(min-width: 808px) 50vw, 100vw"
          style={{
            objectFit: 'cover', // cover, contain, none
          }}
        />
      </div>
      {/* And more images in the grid... */}
    </div>
  )
}
```

### Background Image

<Image
  alt="Background image taking full width and height of page"
  srcLight="/docs/light/background-image.png"
  srcDark="/docs/dark/background-image.png"
  width="1600"
  height="427"
/>

```jsx
import Image from 'next/image'
import mountains from '../public/mountains.jpg'

export default function Background() {
  return (
    <Image
      alt="Mountains"
      src={mountains}
      placeholder="blur"
      quality={100}
      fill
      sizes="100vw"
      style={{
        objectFit: 'cover',
      }}
    />
  )
}
```

For examples of the Image component used with the various styles, see the [Image Component Demo](https://image-component.nextjs.gallery).

## Other Properties

[**View all properties available to the `next/image` component.**](/docs/app/api-reference/components/image)

## Configuration

The `next/image` component and Next.js Image Optimization API can be configured in the [`next.config.js` file](/docs/app/api-reference/next-config-js). These configurations allow you to [enable remote images](/docs/app/api-reference/components/image#remotepatterns), [define custom image breakpoints](/docs/app/api-reference/components/image#devicesizes), [change caching behavior](/docs/app/api-reference/components/image#caching-behavior) and more.

[**Read the full image configuration documentation for more information.**](/docs/app/api-reference/components/image#configuration-options)

```
---
title: Font Optimization
nav_title: Fonts
description: Optimize your application's web fonts with the built-in `next/font` loaders.
related:
  title: API Reference
  description: Learn more about the next/font API.
  links:
    - app/api-reference/components/font
---
```

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

[**`next/font`**](/docs/app/api-reference/components/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)](https://www.youtube.com/watch?v=L8_98i_bMMA).

`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](https://fonts.google.com/

variablefonts) for the best performance and flexibility.

<AppOnly>

```tsx filename="app/layout.tsx" switcher
import { Inter } from 'next/font/google'

// If loading a variable font, you don't need to specify the font weight
const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={inter.className}>
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import { Inter } from 'next/font/google'

// If loading a variable font, you don't need to specify the font weight
const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export default function RootLayout({ children }) {
  return (
    <html lang="en" className={inter.className}>
      <body>{children}</body>
    </html>
  )
}
```

If you can't use a variable font, you will **need to specify a weight**:

```tsx filename="app/layout.tsx" switcher
```

```
import { Roboto } from 'next/font/google'

const roboto = Roboto({
  weight: '400',
  subsets: ['latin'],
  display: 'swap',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={roboto.className}>
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import { Roboto } from 'next/font/google'

const roboto = Roboto({
  weight: '400',
  subsets: ['latin'],
  display: 'swap',
})

export default function RootLayout({ children }) {
  return (
    <html lang="en" className={roboto.className}>
      <body>{children}</body>
    </html>
  )
}
```

</AppOnly>

<PagesOnly>

To use the font in all your pages, add it to [`_app.js` file](/docs/pages/building-your-application/routing/custom-app) under `/pages` as shown below:

```jsx filename="pages/_app.js"
```

import { Inter } from 'next/font/google'

// If loading a variable font, you don't need to specify the font weight
const inter = Inter({ subsets: ['latin'] })

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={inter.className}>
      <Component {...pageProps} />
    </main>
)
}

If you can't use a variable font, you will **need to specify a weight**:

```jsx filename="pages/_app.js"
import { Roboto } from 'next/font/google'

const roboto = Roboto({
  weight: '400',
  subsets: ['latin'],
})

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={roboto.className}>
      <Component {...pageProps} />
    </main>
  )
}
```

</PagesOnly>

You can specify multiple weights and/or styles by using an array:

```jsx filename="app/layout.js"
const roboto = Roboto({
  weight: ['400', '700'],
  style: ['normal', 'italic'],
  subsets: ['latin'],
  display: 'swap',
})
```

> **Good to know**: Use an underscore (_) for font names with multiple

words. E.g. `Roboto Mono` should be imported as `Roboto_Mono`.

<PagesOnly>

### Apply the font in `<head>`

You can also use the font without a wrapper and `className` by injecting it inside the `<head>` as follows:

```jsx filename="pages/_app.js"
import { Inter } from 'next/font/google'

const inter = Inter({ subsets: ['latin'] })

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <style jsx global>{`
        html {
          font-family: ${inter.style.fontFamily};
        }
      `}</style>
      <Component {...pageProps} />
    </>
  )
}
```

### Single page usage

To use the font on a single page, add it to the specific page as shown below:

```jsx filename="pages/index.js"
import { Inter } from 'next/font/google'

const inter = Inter({ subsets: ['latin'] })

export default function Home() {
  return (
    <div className={inter.className}>
      <p>Hello World</p>
    </div>
  )
}
```

</PagesOnly>

### Specifying a subset

Google Fonts are automatically [subset](https://fonts.google.com/knowledge/glossary/subsetting). 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`](/docs/app/api-reference/components/font#preload) is `true` will result in a warning.

This can be done by adding it to the function call:

<AppOnly>

```tsx filename="app/layout.tsx" switcher
const inter = Inter({ subsets: ['latin'] })
```

```jsx filename="app/layout.js" switcher
const inter = Inter({ subsets: ['latin'] })
```

</AppOnly>

<PagesOnly>

```jsx filename="pages/_app.js"
const inter = Inter({ subsets: ['latin'] })
```

</PagesOnly>

View the [Font API Reference](/docs/app/api-reference/components/font) 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 filename="app/fonts.ts" switcher
import { Inter, Roboto_Mono } from 'next/font/google'

export const inter = Inter({
```

~~~
  subsets: ['latin'],
  display: 'swap',
})

export const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
})
```

```js filename="app/fonts.js" switcher
import { Inter, Roboto_Mono } from 'next/font/google'

export const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
})

export const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
})
```

<AppOnly>

```tsx filename="app/layout.tsx" switcher
import { inter } from './fonts'

export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <html lang="en" className={inter.className}>
      <body>
        <div>{children}</div>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import { inter } from './fonts'

export default function Layout({ children }) {
  return (
    <html lang="en" className={inter.className}>
      <body>
~~~

      <div>{children}</div>
    </body>
  </html>
)
}

```tsx filename="app/page.tsx" switcher
import { roboto_mono } from './fonts'

export default function Page() {
  return (
    <>
      <h1 className={roboto_mono.className}>My page</h1>
    </>
  )
}
```

```jsx filename="app/page.js" switcher
import { roboto_mono } from './fonts'

export default function Page() {
  return (
    <>
      <h1 className={roboto_mono.className}>My page</h1>
    </>
  )
}
```

</AppOnly>

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](/docs/app/api-reference/components/font#variable) and use it with your preferred CSS solution:

<AppOnly>

```tsx filename="app/layout.tsx" switcher
import { Inter, Roboto_Mono } from 'next/font/google'
import styles from './global.css'

const inter = Inter({
  subsets: ['latin'],
  variable: '--font-inter',
  display: 'swap',
})

const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  variable: '--font-roboto-mono',
  display: 'swap',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={`${inter.variable} ${roboto_mono.variable}`}>
      <body>
        <h1>My App</h1>
        <div>{children}</div>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import { Inter, Roboto_Mono } from 'next/font/google'

const inter = Inter({
  subsets: ['latin'],
  variable: '--font-inter',
  display: 'swap',
})

const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  variable: '--font-roboto-mono',
  display: 'swap',
})

export default function RootLayout({ children }) {
  return (
    <html lang="en" className={`${inter.variable} ${roboto_mono.variable}`}>
      <body>
        <h1>My App</h1>
        <div>{children}</div>
```

      </body>
    </html>
)
}

</AppOnly>

```css filename="app/global.css"
html {
  font-family: var(--font-inter);
}

h1 {
  font-family: var(--font-roboto-mono);
}
```

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](https://fonts.google.com/variablefonts) for the best performance and flexibility.

<AppOnly>

```tsx filename="app/layout.tsx" switcher
import localFont from 'next/font/local'

// Font files can be colocated inside of `app`
const myFont = localFont({
  src: './my-font.woff2',
  display: 'swap',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
```

    <html lang="en" className={myFont.className}>
      <body>{children}</body>
    </html>
)
}

```jsx filename="app/layout.js" switcher
import localFont from 'next/font/local'

// Font files can be colocated inside of `app`
const myFont = localFont({
  src: './my-font.woff2',
  display: 'swap',
})

export default function RootLayout({ children }) {
  return (
    <html lang="en" className={myFont.className}>
      <body>{children}</body>
    </html>
  )
}
```

</AppOnly>

<PagesOnly>

```jsx filename="pages/_app.js"
import localFont from 'next/font/local'

// Font files can be colocated inside of `pages`
const myFont = localFont({ src: './my-font.woff2' })

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={myFont.className}>
      <Component {...pageProps} />
    </main>
  )
}
```

</PagesOnly>

If you want to use multiple files for a single font family, `src` can be an array:

```js
const roboto = localFont({
  src: [
    {
      path: './Roboto-Regular.woff2',
      weight: '400',
      style: 'normal',
    },
    {
      path: './Roboto-Italic.woff2',
      weight: '400',
      style: 'italic',
    },
    {
      path: './Roboto-Bold.woff2',
      weight: '700',
      style: 'normal',
    },
    {
      path: './Roboto-BoldItalic.woff2',
      weight: '700',
      style: 'italic',
    },
  ],
})
```

View the [Font API Reference](/docs/app/api-reference/components/font) for more information.

## With Tailwind CSS

`next/font` can be used with [Tailwind CSS](https://tailwindcss.com/) through a [CSS variable](/docs/app/api-reference/components/font#css-variables).

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.

<AppOnly>

```tsx filename="app/layout.tsx" switcher
import { Inter, Roboto_Mono } from 'next/font/google'

const inter = Inter({
```

~~~
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-inter',
})

const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-roboto-mono',
})

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en" className={`${inter.variable} ${roboto_mono.variable}`}>
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import { Inter, Roboto_Mono } from 'next/font/google'

const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-inter',
})

const roboto_mono = Roboto_Mono({
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-roboto-mono',
})

export default function RootLayout({ children }) {
  return (
    <html lang="en" className={`${inter.variable} ${roboto_mono.variable}`}>
      <body>{children}</body>
    </html>
  )
}
```
~~~

</AppOnly>

<PagesOnly>

```jsx filename="pages/_app.js"
import { Inter } from 'next/font/google'

const inter = Inter({
  subsets: ['latin'],
  variable: '--font-inter',
})

export default function MyApp({ Component, pageProps }) {
  return (
    <main className={`${inter.variable} font-sans`}>
      <Component {...pageProps} />
    </main>
  )
}
```

</PagesOnly>

Finally, add the CSS variable to your [Tailwind CSS config](/docs/app/building-your-application/styling/tailwind-css#configuring-tailwind):

```js filename="tailwind.config.js"
/** @type {import('tailwindcss').Config} */
module.exports = {
  content: [
    './pages/**/*.{js,ts,jsx,tsx}',
    './components/**/*.{js,ts,jsx,tsx}',
    './app/**/*.{js,ts,jsx,tsx}',
  ],
  theme: {
    extend: {
      fontFamily: {
        sans: ['var(--font-inter)'],
        mono: ['var(--font-roboto-mono)'],
      },
    },
  },
  plugins: [],
}
```

You can now use the `font-sans` and `font-mono` utility classes to apply the font to your elements.

## Preloading

<AppOnly>
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](/docs/app/building-your-application/routing/pages-and-layouts#pages), it is preloaded on the unique route for that page.
- If it's a [layout](/docs/app/building-your-application/routing/pages-and-layouts#layouts), it is preloaded on all the routes wrapped by the layout.
- If it's the [root layout](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required), it is preloaded on all routes.

</AppOnly>

<PagesOnly>

When a font function is called on a page of your site, it is not globally available and preloaded on all routes. Rather, the font is only preloaded on the related route/s based on the type of file where it is used:

- if it's a [unique page](/docs/pages/building-your-application/routing/pages-and-layouts), it is preloaded on the unique route for that page
- if it's in the [custom App](/docs/pages/building-your-application/routing/custom-app), it is preloaded on all the routes of the site under `/pages`

</PagesOnly>

## 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

---
title: Script Optimization
nav_title: Scripts

description: Optimize 3rd party scripts with the built-in Script component.
related:
  title: API Reference
  description: Learn more about the next/script API.
  links:
    - app/api-reference/components/script

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<AppOnly>

### Layout Scripts

To load a third-party script for multiple routes, import `next/script` and include the script directly in your layout component:

```tsx filename="app/dashboard/layout.tsx" switcher
import Script from 'next/script'

export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <>
      <section>{children}</section>
      <Script src="https://example.com/script.js" />
    </>
  )
}
```

```jsx filename="app/dashboard/layout.js" switcher
import Script from 'next/script'

export default function DashboardLayout({ children }) {
  return (
    <>
      <section>{children}</section>
      <Script src="https://example.com/script.js" />
    </>
  )
```

}

The third-party script is fetched when 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.

</AppOnly>

### Application Scripts

<AppOnly>

To load a third-party script for all routes, import `next/script` and include the script directly in your root layout:

```tsx filename="app/layout.tsx" switcher
import Script from 'next/script'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
      <Script src="https://example.com/script.js" />
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import Script from 'next/script'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
      <Script src="https://example.com/script.js" />
    </html>
  )
}
```

</AppOnly>

<PagesOnly>

To load a third-party script for all routes, import `next/script` and include the script directly in your custom `_app`:

```jsx filename="pages/_app.js"
import Script from 'next/script'

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <Component {...pageProps} />
      <Script src="https://example.com/script.js" />
    </>
  )
}
```

</PagesOnly>

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 to 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`](/docs/app/api-reference/components/script#strategy) 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`](/docs/app/building-your-application/routing/defining-routes) directory. Use with caution.

Scripts that use the `worker` strategy are offloaded and executed in a web worker with [Partytown](https://partytown.builder.io/). This can improve the performance of your site by dedicating the main thread to the rest of your application code.

This strategy is still experimental and can only be used if the `nextScriptWorkers` flag is enabled in `next.config.js`:

```js filename="next.config.js"
module.exports = {
  experimental: {
    nextScriptWorkers: true,
  },
}
```

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:

```bash filename="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.

```tsx filename="pages/home.tsx" switcher
import Script from 'next/script'

export default function Home() {
  return (
    <>
      <Script src="https://example.com/script.js" strategy="worker" />
    </>
  )
}
```

```jsx filename="pages/home.js" switcher
import Script from 'next/script'

export default function Home() {
  return (
    <>
      <Script src="https://example.com/script.js" strategy="worker" />
    </>
  )
}
```

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](https://partytown.builder.io/trade-offs) 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:

```jsx
<Script id="show-banner">
  {`document.getElementById('banner').classList.remove('hidden')`}
</Script>
```

Or by using the `dangerouslySetInnerHTML` property:

```jsx
<Script
  id="show-banner"
  dangerouslySetInnerHTML={{
    __html: `document.getElementById('banner').classList.remove('hidden')`,
  }}
/>
```

> **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.

<AppOnly>

These handlers will only work when `next/script` is imported and used inside of a [Client Component](/docs/app/building-your-application/rendering/client-components) where `"use client"` is defined as the first line of code:

```tsx filename="app/page.tsx" switcher
'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onLoad={() => {
          console.log('Script has loaded')
        }}
      />
    </>
  )
}
```

```jsx filename="app/page.js" switcher
'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onLoad={() => {
          console.log('Script has loaded')
        }}
      />
    </>
  )
```

~~~
}
```

Refer to the [`next/script`](/docs/app/api-reference/components/script#onload) API reference to learn more about each event handler and view examples.

</AppOnly>

<PagesOnly>

These handlers will only work when `next/script` is imported and used inside of a [Client Component](/docs/app/building-your-application/rendering/client-components) where `"use client"` is defined as the first line of code:

```tsx filename="pages/index.tsx" switcher
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onLoad={() => {
          console.log('Script has loaded')
        }}
      />
    </>
  )
}
```

```jsx filename="pages/index.js" switcher
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onLoad={() => {
          console.log('Script has loaded')
        }}
      />
    </>
  )
}
~~~

Refer to the [`next/script`](/docs/pages/api-reference/components/script#onload) API reference to learn more about each event handler and view examples.

</PagesOnly>

### 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`](https://developer.mozilla.org/docs/Web/HTML/Global_attributes/nonce) or [custom data attributes](https://developer.mozilla.org/docs/Web/HTML/Global_attributes/data-*). Including any additional attributes will automatically forward it to the final, optimized `<script>` element that is included in the HTML.

<AppOnly>

```tsx filename="app/page.tsx" switcher
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        id="example-script"
        nonce="XUENAJFW"
        data-test="script"
      />
    </>
  )
}
```

```jsx filename="app/page.js" switcher
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        id="example-script"
        nonce="XUENAJFW"
```

        data-test="script"
      />
    </>
)
}

</AppOnly>

<PagesOnly>

```tsx filename="pages/index.tsx" switcher
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        id="example-script"
        nonce="XUENAJFW"
        data-test="script"
      />
    </>
  )
}
```

```jsx filename="pages/index.js" switcher
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        id="example-script"
        nonce="XUENAJFW"
        data-test="script"
      />
    </>
  )
}
```

</PagesOnly>

---
title: Metadata
description: Use the Metadata API to define metadata in any layout or page.
related:
  description: View all the Metadata API options.
  links:
    - app/api-reference/functions/generate-metadata
    - app/api-reference/file-conventions/metadata
    - app/api-reference/functions/generate-viewport

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](/docs/app/api-reference/functions/generate-metadata#metadata-object) or a dynamic [`generateMetadata` function](/docs/app/api-reference/functions/generate-metadata#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`](/docs/app/api-reference/functions/image-response) constructor.

## Static Metadata

To define static metadata, export a [`Metadata` object](/docs/app/api-reference/functions/generate-metadata#metadata-object) from a `layout.js` or static `page.js` file.

```tsx filename="layout.tsx | page.tsx" switcher
import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: '...',
  description: '...',
}

export default function Page() {}
```

```jsx filename="layout.js | page.js" switcher
export const metadata = {
  title: '...',
  description: '...',
}

export default function Page() {}
```

For all the available options, see the [API Reference](/docs/app/api-reference/functions/generate-metadata).

## Dynamic Metadata

You can use `generateMetadata` function to `fetch` metadata that requires dynamic values.

```tsx filename="app/products/[id]/page.tsx" switcher
import type { Metadata, ResolvingMetadata } from 'next'

type Props = {
  params: { id: string }
  searchParams: { [key: string]: string | string[] | undefined }
}

export async function generateMetadata(
  { params, searchParams }: Props,
  parent: ResolvingMetadata
): Promise<Metadata> {
  // read route params
  const id = params.id

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}
```

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

```jsx filename="app/products/[id]/page.js" switcher
export async function generateMetadata({ params, searchParams }, parent) {
  // read route params
  const id = params.id

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}

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

For all the available params, see the [API Reference](/docs/app/api-reference/functions/generate-metadata).

> **Good to know**:
>
> - Both static and dynamic metadata through `generateMetadata` are **only supported in Server Components**.
> - `fetch` requests are automatically [memoized](/docs/app/building-your-application/caching#request-memoization) for the same data across `generateMetadata`, `generateStaticParams`, Layouts, Pages, and Server Components. React [`cache` can be used](/docs/app/building-your-application/caching#request-memoization) 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](/docs/app/building-your-application/routing/loading-ui-and-streaming) includes `<head>` tags.

## File-based metadata

These special files are available for metadata:

- [favicon.ico, apple-icon.jpg, and icon.jpg](/docs/app/api-reference/file-conventions/metadata/app-icons)
- [opengraph-image.jpg and twitter-image.jpg](/docs/app/api-reference/file-conventions/metadata/opengraph-image)
- [robots.txt](/docs/app/api-reference/file-conventions/metadata/robots)
- [sitemap.xml](/docs/app/api-reference/file-conventions/metadata/sitemap)

You can use these for static metadata, or you can programmatically generate these files with code.

For implementation and examples, see the [Metadata Files](/docs/app/api-reference/file-conventions/metadata) API Reference and [Dynamic Image Generation](#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](https://developer.mozilla.org/docs/Web/HTML/Element/meta#attr-charset) sets the character encoding for the website.
- The [meta viewport tag](https://developer.mozilla.org/docs/Web/HTML/Viewport_meta_tag) sets the viewport width and scale for the website to adjust for different devices.

```html
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

> **Good to know**: You can overwrite the default [`viewport`](/docs/app/api-reference/functions/generate-metadata#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](#ordering), 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`](/docs/app/api-reference/functions/generate-metadata#opengraph) and [`robots`](/docs/app/api-reference/functions/generate-metadata#robots) that are defined in an earlier segment are **overwritten** by the last segment to define them.

#### Overwriting fields

```jsx filename="app/layout.js"
export const metadata = {
  title: 'Acme',
  openGraph: {
    title: 'Acme',
    description: 'Acme is a...',
  },
}
```

```jsx filename="app/blog/page.js"
export const metadata = {
  title: 'Blog',
  openGraph: {
    title: 'Blog',
  },
}

// Output:
// <title>Blog</title>
// <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:

```jsx filename="app/shared-metadata.js"
export const openGraphImage = { images: ['http://...'] }
```

```jsx filename="app/page.js"
import { openGraphImage } from './shared-metadata'

export const metadata = {
  openGraph: {
    ...openGraphImage,
    title: 'Home',
  },
}
```

```jsx filename="app/about/page.js"
import { openGraphImage } from '../shared-metadata'

export const metadata = {
  openGraph: {
    ...openGraphImage,
    title: 'About',
  },
}
```

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

```jsx filename="app/layout.js"
export const metadata = {
  title: 'Acme',
  openGraph: {
    title: 'Acme',
    description: 'Acme is a...',
  },
}
```

```jsx filename="app/about/page.js"
export const metadata = {
  title: 'About',
}
```

```
// Output:
// <title>About</title>
// <meta property="og:title" content="Acme" />
// <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](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes#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/og`:

```jsx filename="app/about/route.js"
import { ImageResponse } from 'next/og'

export const runtime = 'edge'

export async function GET() {
  return new ImageResponse(
    (
      <div
        style={{
          fontSize: 128,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          textAlign: 'center',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        Hello world!
      </div>
    ),
    {
      width: 1200,
      height: 600,
    }
  )
}
```

`ImageResponse` integrates well with other Next.js APIs, including [Route Handlers](/docs/app/building-your-application/routing/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](/docs/app/api-reference/functions/image-response).

> **Good to know**:
>
> - Examples are available in the [Vercel OG Playground](https://og-playground.vercel.app/).
> - `ImageResponse` uses [@vercel/og](https://vercel.com/docs/concepts/functions/edge-functions/og-image-generation), [Satori](https://github.com/vercel/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](https://json-ld.org/) 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:

```tsx filename="app/products/[id]/page.tsx" switcher
export default async function Page({ params }) {
  const product = await getProduct(params.id)

  const jsonLd = {
    '@context': 'https://schema.org',
    '@type': 'Product',
    name: product.name,
    image: product.image,
    description: product.description,
  }

  return (
    <section>
      {/* Add JSON-LD to your page */}
      <script
        type="application/ld+json"
        dangerouslySetInnerHTML={{ __html: JSON.stringify(jsonLd) }}
      />
      {/* ... */}
    </section>
  )
}
```

```jsx filename="app/products/[id]/page.js" switcher
export default async function Page({ params }) {
  const product = await getProduct(params.id)

  const jsonLd = {
    '@context': 'https://schema.org',
    '@type': 'Product',
    name: product.name,
    image: product.image,
    description: product.description,
  }

  return (
    <section>
      {/* Add JSON-LD to your page */}
      <script
        type="application/ld+json"
        dangerouslySetInnerHTML={{ __html: JSON.stringify(jsonLd) }}
      />
      {/* ... */}
    </section>
```

)
}

You can validate and test your structured data with the [Rich Results Test](https://search.google.com/test/rich-results) for Google or the generic [Schema Markup Validator](https://validator.schema.org/).

You can type your JSON-LD with TypeScript using community packages like [`schema-dts`](https://www.npmjs.com/package/schema-dts):

```tsx
import { Product, WithContext } from 'schema-dts'

const jsonLd: WithContext<Product> = {
  '@context': 'https://schema.org',
  '@type': 'Product',
  name: 'Next.js Sticker',
  image: 'https://nextjs.org/imgs/sticker.png',
  description: 'Dynamic at the speed of static.',
}
```

---
title: Static Assets
description: Next.js allows you to serve static files, like images, in the public directory. You can learn how it works here.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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:

```jsx filename="Avatar.js"
import Image from 'next/image'

export function Avatar() {
  return <Image src="/me.png" alt="me" width="64" height="64" />
}
```

```

<PagesOnly>

This folder is also useful for `robots.txt`, `favicon.ico`, Google Site Verification, and any other static files (including `.html`). But make 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](/docs/messages/conflicting-public-file-page).

</PagesOnly>

<AppOnly>

For static metadata files, such as `robots.txt`, `favicon.ico`, etc, you should use [special metadata files](/docs/app/api-reference/file-conventions/metadata) inside the `app` folder.

</AppOnly>

> Good to know:
>
> - The directory must be named `public`. The name cannot be changed and it's the only directory used to serve static assets.
> - Only assets that are in the `public` directory at [build time](/docs/app/api-reference/next-cli#build) will be served by Next.js. Files added at request time won't be available. We recommend using a third-party service like [AWS S3](https://aws.amazon.com/s3/) for persistent file storage.

---
title: Lazy Loading
description: Lazy load imported libraries and React Components to improve your application's loading performance.
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

[Lazy loading](https://developer.mozilla.org/docs/Web/Performance/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](#nextdynamic) with `next/dynamic`
2. Using [`React.lazy()`](https://react.dev/reference/react/lazy) with [Suspense](https://react.dev/reference/react/Suspense)

By default, Server Components are automatically [code split](https://developer.mozilla.org/docs/Glossary/Code_splitting), and you can use [streaming](/docs/app/building-your-application/routing/loading-ui-and-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()`](https://react.dev/reference/react/lazy) and [Suspense](https://react.dev/reference/react/Suspense). It behaves the same way in the `app` and `pages` directories to allow for incremental migration.

## Examples

<AppOnly>
### Importing Client Components

```jsx filename="app/page.js"
'use client'

import { useState } from 'react'
import dynamic from 'next/dynamic'

// Client Components:
const ComponentA = dynamic(() => import('../components/A'))
const ComponentB = dynamic(() => import('../components/B'))
const ComponentC = dynamic(() => import('../components/C'), { ssr: false })

export default function ClientComponentExample() {
  const [showMore, setShowMore] = useState(false)

  return (
    <div>
      {/* Load immediately, but in a separate client bundle */}
      <ComponentA />

      {/* Load on demand, only when/if the condition is met */}
      {showMore && <ComponentB />}
      <button onClick={() => setShowMore(!showMore)}>Toggle</button>
```

    {/* Load only on the client side */}
    <ComponentC />
  </div>
)
}

### Skipping 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`:

```jsx
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.

```jsx filename="app/page.js"
import dynamic from 'next/dynamic'

// Server Component:
const ServerComponent = dynamic(() => import('../components/
ServerComponent'))

export default function ServerComponentExample() {
 return (
  <div>
    <ServerComponent />
  </div>
 )
}
```

### Loading External Libraries

External libraries can be loaded on demand using the [`import()`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/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.

```jsx filename="app/page.js"
'use client'

import { useState } from 'react'

const names = ['Tim', 'Joe', 'Bel', 'Lee']

export default function Page() {
  const [results, setResults] = useState()

  return (
    <div>
      <input
        type="text"
        placeholder="Search"
        onChange={async (e) => {
          const { value } = e.currentTarget
          // Dynamically load fuse.js
          const Fuse = (await import('fuse.js')).default
          const fuse = new Fuse(names)

          setResults(fuse.search(value))
        }}
      />
      <pre>Results: {JSON.stringify(results, null, 2)}</pre>
    </div>
  )
}
```

### Adding a custom loading component

```jsx filename="app/page.js"
import dynamic from 'next/dynamic'

const WithCustomLoading = dynamic(
  () => import('../components/WithCustomLoading'),
  {
    loading: () => <p>Loading...</p>,
  }
)

export default function Page() {
  return (
    <div>
```

      {/* The loading component will be rendered while <WithCustomLoading/> is loading */}
      <WithCustomLoading />
    </div>
)
}

### Importing Named Exports

To dynamically import a named export, you can return it from the Promise returned by [`import()`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/import) function:

```jsx filename="components/hello.js"
'use client'

export function Hello() {
  return <p>Hello!</p>
}
```

```jsx filename="app/page.js"
import dynamic from 'next/dynamic'

const ClientComponent = dynamic(() =>
  import('../components/hello').then((mod) => mod.Hello)
)
```

</AppOnly>

<PagesOnly>

By using `next/dynamic`, the header component will not be included in the page's initial JavaScript bundle. The page will render the Suspense `fallback` first, followed by the `Header` component when the `Suspense` boundary is resolved.

```jsx
import dynamic from 'next/dynamic'

const DynamicHeader = dynamic(() => import('../components/header'), {
  loading: () => <p>Loading...</p>,
})

export default function Home() {
```

  return <DynamicHeader />
}

> **Good to know**: In `import('path/to/component')`, the path must be explicitly written. It can't be a template string nor a variable. Furthermore the `import()` has to be inside the `dynamic()` call for Next.js to be able to match webpack bundles / module ids to the specific `dynamic()` call and preload them before rendering. `dynamic()` can't be used inside of React rendering as it needs to be marked in the top level of the module for preloading to work, similar to `React.lazy`.

## With named exports

To dynamically import a named export, you can return it from the [Promise](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Promise) returned by [`import()`](https://github.com/tc39/proposal-dynamic-import#example):

```jsx filename="components/hello.js"
export function Hello() {
  return <p>Hello!</p>
}

// pages/index.js
import dynamic from 'next/dynamic'

const DynamicComponent = dynamic(() =>
  import('../components/hello').then((mod) => mod.Hello)
)
```

## With no SSR

To dynamically load a component on the client side, you can use the `ssr` option to disable server-rendering. This is useful if an external dependency or component relies on browser APIs like `window`.

```jsx
import dynamic from 'next/dynamic'

const DynamicHeader = dynamic(() => import('../components/header'), {
  ssr: false,
})
```

## With external libraries

This example uses the external library `fuse.js` for fuzzy search. The module is only loaded in the browser after the user types in the search input.

```jsx
import { useState } from 'react'

const names = ['Tim', 'Joe', 'Bel', 'Lee']

export default function Page() {
  const [results, setResults] = useState()

  return (
    <div>
      <input
        type="text"
        placeholder="Search"
        onChange={async (e) => {
          const { value } = e.currentTarget
          // Dynamically load fuse.js
          const Fuse = (await import('fuse.js')).default
          const fuse = new Fuse(names)

          setResults(fuse.search(value))
        }}
      />
      <pre>Results: {JSON.stringify(results, null, 2)}</pre>
    </div>
  )
}
```

</PagesOnly>

---
title: Analytics
description: Measure and track page performance using Next.js Speed Insights
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js has built-in support for measuring and reporting performance metrics. You can either use the `useReportWebVitals` hook to manage reporting yourself, or alternatively, Vercel provides a [managed service](https://

vercel.com/analytics?utm_source=next-site&utm_medium=docs&utm_campaign=next-website) to automatically collect and visualize metrics for you.

## Build Your Own

<PagesOnly>

```jsx filename="pages/_app.js"
import { useReportWebVitals } from 'next/web-vitals'

function MyApp({ Component, pageProps }) {
  useReportWebVitals((metric) => {
    console.log(metric)
  })

  return <Component {...pageProps} />
}
```

View the [API Reference](/docs/pages/api-reference/functions/use-report-web-vitals) for more information.

</PagesOnly>

<AppOnly>

```jsx filename="app/_components/web-vitals.js"
'use client'

import { useReportWebVitals } from 'next/web-vitals'

export function WebVitals() {
  useReportWebVitals((metric) => {
    console.log(metric)
  })
}
```

```jsx filename="app/layout.js"
import { WebVitals } from './_components/web-vitals'

export default function Layout({ children }) {
  return (
    <html>
      <body>
        <WebVitals />
```

```jsx
      {children}
    </body>
  </html>
 )
}
```

> 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.

View the [API Reference](/docs/app/api-reference/functions/use-report-web-vitals) for more information.

</AppOnly>

## Web Vitals

[Web Vitals](https://web.dev/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](https://developer.mozilla.org/docs/Glossary/Time_to_first_byte) (TTFB)
- [First Contentful Paint](https://developer.mozilla.org/docs/Glossary/First_contentful_paint) (FCP)
- [Largest Contentful Paint](https://web.dev/lcp/) (LCP)
- [First Input Delay](https://web.dev/fid/) (FID)
- [Cumulative Layout Shift](https://web.dev/cls/) (CLS)
- [Interaction to Next Paint](https://web.dev/inp/) (INP)

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

<PagesOnly>

```jsx filename="pages/_app.js"
import { useReportWebVitals } from 'next/web-vitals'

function MyApp({ Component, pageProps }) {
 useReportWebVitals((metric) => {
  switch (metric.name) {
   case 'FCP': {
     // handle FCP results
   }
   case 'LCP': {
     // handle LCP results
```

```
      }
      // ...
    }
  })

  return <Component {...pageProps} />
}
```

</PagesOnly>

<AppOnly>

```tsx filename="app/components/web-vitals.tsx" switcher
'use client'

import { useReportWebVitals } from 'next/web-vitals'

export function WebVitals() {
  useReportWebVitals((metric) => {
    switch (metric.name) {
      case 'FCP': {
        // handle FCP results
      }
      case 'LCP': {
        // handle LCP results
      }
      // ...
    }
  })
}
```

```jsx filename="app/components/web-vitals.js" switcher
'use client'

import { useReportWebVitals } from 'next/web-vitals'

export function WebVitals() {
  useReportWebVitals((metric) => {
    switch (metric.name) {
      case 'FCP': {
        // handle FCP results
      }
      case 'LCP': {
        // handle LCP results
      }
      // ...
    }
  })
}
```

</AppOnly>

<PagesOnly>

## Custom Metrics

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

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

You can handle all the results of these metrics separately:

```js
export function reportWebVitals(metric) {
  switch (metric.name) {
    case 'Next.js-hydration':
      // handle hydration results
      break
    case 'Next.js-route-change-to-render':
      // handle route-change to render results
      break
    case 'Next.js-render':
      // handle render results
      break
    default:
      break
  }
}
```

These metrics work in all browsers that support the [User Timing API](https://caniuse.com/#feat=user-timing).

</PagesOnly>

## Sending results to external systems

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

```js
useReportWebVitals((metric) => {
  const body = JSON.stringify(metric)
  const url = 'https://example.com/analytics'

  // Use `navigator.sendBeacon()` if available, falling back to `fetch()`.
  if (navigator.sendBeacon) {
    navigator.sendBeacon(url, body)
  } else {
    fetch(url, { body, method: 'POST', keepalive: true })
  }
})
```

> **Good to know**: If you use [Google Analytics](https://analytics.google.com/analytics/web/), using the
> `id` value can allow you to construct metric distributions manually (to calculate percentiles,
> etc.)

> ```js
> useReportWebVitals(metric => {
>   // Use `window.gtag` if you initialized Google Analytics as this example:
>   // https://github.com/vercel/next.js/blob/canary/examples/with-google-analytics/pages/_app.js
>   window.gtag('event', metric.name, {
>     value: Math.round(metric.name === 'CLS' ? metric.value * 1000 : metric.value), // values must be integers
>     event_label: metric.id, // id unique to current page load
>     non_interaction: true, // avoids affecting bounce rate.
>   });
> }
> ```
>
> Read more about [sending results to Google Analytics](https://github.com/GoogleChrome/web-vitals#send-the-results-to-google-analytics).

---
title: OpenTelemetry

description: Learn how to instrument your Next.js app with OpenTelemetry.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

> **Good to know**: 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](https://opentelemetry.io/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](https://opentelemetry.io/docs/concepts/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.

> **Good to know**: 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 if you need to customize your setup.

### Using `@vercel/otel`

To get started, you must install `@vercel/otel`:

```bash filename="Terminal"
npm install @vercel/otel
```

<AppOnly>

Next, create a custom [`instrumentation.ts`](/docs/app/building-your-application/optimizing/instrumentation) (or `.js`) file in the **root directory** of the project (or inside `src` folder if using one):

</AppOnly>

<PagesOnly>

Next, create a custom [`instrumentation.ts`](/docs/pages/building-your-application/optimizing/instrumentation) (or `.js`) file in the **root directory** of the project (or inside `src` folder if using one):

</PagesOnly>

```ts filename="your-project/instrumentation.ts" switcher
import { registerOTel } from '@vercel/otel'

export function register() {
  registerOTel('next-app')
}
```

```js filename="your-project/instrumentation.js" switcher
import { registerOTel } from '@vercel/otel'

export function register() {
  registerOTel('next-app')
}
```

<AppOnly>

> **Good to know**

>
> - The `instrumentation` file should be in the root of your project and not inside the `app` or `pages` directory. If you're using the `src` folder, then place the file inside `src` alongside `pages` and `app`.
> - If you use the [`pageExtensions` config option](/docs/app/api-reference/next-config-js/pageExtensions) to add a suffix, you will also need to update the `instrumentation` filename to match.
> - We have created a basic [with-opentelemetry](https://github.com/vercel/next.js/tree/canary/examples/with-opentelemetry) example that you can use.

</AppOnly>

<PagesOnly>

> **Good to know**
>
> - The `instrumentation` file should be in the root of your project and not inside the `app` or `pages` directory. If you're using the `src` folder, then place the file inside `src` alongside `pages` and `app`.
> - If you use the [`pageExtensions` config option](/docs/pages/api-reference/next-config-js/pageExtensions) to add a suffix, you will also need to update the `instrumentation` filename to match.
> - We have created a basic [with-opentelemetry](https://github.com/vercel/next.js/tree/canary/examples/with-opentelemetry) example that you can use.

</PagesOnly>

### 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:

```bash filename="Terminal"
npm install @opentelemetry/sdk-node @opentelemetry/resources @opentelemetry/semantic-conventions @opentelemetry/sdk-trace-node @opentelemetry/exporter-trace-otlp-http
```

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 filename="instrumentation.ts" switcher
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation.node.ts')
  }
}
```

```js filename="instrumentation.js" switcher
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation.node.js')
  }
}
```

```ts filename="instrumentation.node.ts" switcher
import { NodeSDK } from '@opentelemetry/sdk-node'
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http'
import { Resource } from '@opentelemetry/resources'
import { SemanticResourceAttributes } from '@opentelemetry/semantic-conventions'
import { SimpleSpanProcessor } from '@opentelemetry/sdk-trace-node'

const sdk = new NodeSDK({
  resource: new Resource({
    [SemanticResourceAttributes.SERVICE_NAME]: 'next-app',
  }),
  spanProcessor: new SimpleSpanProcessor(new OTLPTraceExporter()),
})
sdk.start()
```

```js filename="instrumentation.node.js" switcher
import { NodeSDK } from '@opentelemetry/sdk-node'
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http'
import { Resource } from '@opentelemetry/resources'
import { SemanticResourceAttributes } from '@opentelemetry/semantic-conventions'
import { SimpleSpanProcessor } from '@opentelemetry/sdk-trace-node'

const sdk = new NodeSDK({
  resource: new Resource({
    [SemanticResourceAttributes.SERVICE_NAME]: 'next-app',
  }),
  spanProcessor: new SimpleSpanProcessor(new OTLPTraceExporter()),
})
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](https://github.com/vercel/opentelemetry-collector-dev-setup).

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](https://vercel.com/docs/concepts/observability/otel-overview/quickstart) 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](https://opentelemetry.io/docs/collector/getting-started/), 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](/docs/pages/building-your-application/optimizing/open-telemetry#manual-opentelemetry-configuration)

## Custom Spans

You can add a custom span with [OpenTelemetry APIs](https://opentelemetry.io/docs/instrumentation/js/instrumentation).

```bash filename="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:

```ts
import { trace } from '@opentelemetry/api'

export async function fetchGithubStars() {
  return await trace
    .getTracer('nextjs-example')
    .startActiveSpan('fetchGithubStars', async (span) => {
      try {
        return await getValue()
      } finally {
        span.end()
      }
    })
}
```

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](https://opentelemetry.io/docs/reference/specification/trace/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](https://opentelemetry.io/docs/reference/specification/trace/semantic_conventions/http/#common-attributes)
  - `http.method`
  - `http.status_code`
- [Server HTTP attributes](https://opentelemetry.io/docs/reference/specification/trace/semantic_conventions/http/#http-server-semantic-conventions)
  - `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](https://opentelemetry.io/docs/reference/specification/trace/semantic_conventions/http/#common-attributes)
  - `http.method`
- [Client HTTP attributes](https://opentelemetry.io/docs/reference/specification/trace/semantic_conventions/http/#http-client)
  - `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`

---
title: Instrumentation
description: Learn how to use instrumentation to run code at server startup in your Next.js app

---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

If you export a function named `register` from a `instrumentation.ts` (or `.js`) file in the **root directory** of your project (or inside the `src` folder if using one), we will call that function whenever a new Next.js server instance is bootstrapped.

<AppOnly>

> **Good to know**
>
> - This feature is **experimental**. To use it, you must explicitly opt in by defining `experimental.instrumentationHook = true;` in your `next.config.js`.
> - The `instrumentation` file should be in the root of your project and not inside the `app` or `pages` directory. If you're using the `src` folder, then place the file inside `src` alongside `pages` and `app`.
> - If you use the [`pageExtensions` config option](/docs/app/api-reference/next-config-js/pageExtensions) to add a suffix, you will also need to update the `instrumentation` filename to match.
> - We have created a basic [with-opentelemetry](https://github.com/vercel/next.js/tree/canary/examples/with-opentelemetry) example that you can use.

</AppOnly>

<PagesOnly>

> **Good to know**
>
> - This feature is **experimental**. To use it, you must explicitly opt in by defining `experimental.instrumentationHook = true;` in your `next.config.js`.
> - The `instrumentation` file should be in the root of your project and not inside the `app` or `pages` directory. If you're using the `src` folder, then place the file inside `src` alongside `pages` and `app`.
> - If you use the [`pageExtensions` config option](/docs/pages/api-reference/next-config-js/pageExtensions) to add a suffix, you will also need to update the `instrumentation` filename to match.
> - We have created a basic [with-opentelemetry](https://github.com/vercel/next.js/tree/canary/examples/with-opentelemetry) example that you can use.

</PagesOnly>

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 filename="your-project/instrumentation.ts" switcher
import { init } from 'package-init'

export function register() {
  init()
}
```

```js filename="your-project/instrumentation.js" switcher
import { init } from 'package-init'

export function register() {
  init()
}
```

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 filename="your-project/instrumentation.ts" switcher
export async function register() {
  await import('package-with-side-effect')
}
```

```js filename="your-project/instrumentation.js" switcher
export async function register() {
  await import('package-with-side-effect')
}
```

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 filename="your-project/instrumentation.ts" switcher
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation-node')
  }

  if (process.env.NEXT_RUNTIME === 'edge') {
    await import('./instrumentation-edge')
  }
}
```

```js filename="your-project/instrumentation.js" switcher
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation-node')
  }

  if (process.env.NEXT_RUNTIME === 'edge') {
    await import('./instrumentation-edge')
  }
}
```

---
title: Third Party Libraries
description: Optimize the performance of third-party libraries in your application with the `@next/third-parties` package.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

`@next/third-parties` is a library that provides a collection of components and utilities that
improve the performance and developer experience of loading popular third-party libraries in your
Next.js application.

> **Good to know**: `@next/third-parties` is a new **experimental** library that is still under

> active development. We're currently working on adding more third-party integrations.

All third-party integrations provided by `@next/third-parties` have been optimized for performance
and ease of use.

## Getting Started

To get started, you must install the `@next/third-parties` library:

```bash filename="Terminal"
npm install @next/third-parties
```

## Google Third-Parties

All supported third-party libraries from Google can be imported from `@next/third-parties/google`.

### Google Tag Manager

The `GoogleTagManager` component can be used to instantiate a [Google Tag Manager](https://developers.google.com/tag-platform/tag-manager) container to your
page. By default, it fetches the original inline script after hydration occurs on the page.

<AppOnly>

To load Google Tag Manager for all routes, include the component directly in your root layout:

```tsx filename="app/layout.tsx" switcher
import { GoogleTagManager } from '@next/third-parties/google'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
      <GoogleTagManager gtmId="GTM-XYZ" />
    </html>
  )
```

}

```jsx filename="app/layout.js" switcher
import { GoogleTagManager } from '@next/third-parties/google'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
      <GoogleTagManager gtmId="GTM-XYZ" />
    </html>
  )
}
```

</AppOnly>

<PagesOnly>

To load Google Tag Manager for all routes, include the component directly in your custom `_app`:

```jsx filename="pages/_app.js"
import { GoogleTagManager } from '@next/third-parties/google'

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <Component {...pageProps} />
      <GoogleTagManager gtmId="GTM-XYZ" />
    </>
  )
}
```

</PagesOnly>

To load Google Tag Manager for a single route, include the component in your page file:

<AppOnly>

```jsx filename="app/page.js"
import { GoogleTagManager } from '@next/third-parties/google'

export default function Page() {
  return <GoogleTagManager gtmId="GTM-XYZ" />
}
```

</AppOnly>

<PagesOnly>

```jsx filename="pages/index.js"
import { GoogleTagManager } from '@next/third-parties/google'

export default function Page() {
  return <GoogleTagManager gtmId="GTM-XYZ" />
}
```

</PagesOnly>

#### Sending Events

The `sendGTMEvent` function can be used to track user interactions on your page by sending events
using the `dataLayer` object. For this function to work, the `<GoogleTagManager />` component must be
included in either a parent layout, page, or component, or directly in the same file.

<AppOnly>

```jsx filename="app/page.js"
'use client'

import { sendGTMEvent } from '@next/third-parties/google'

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGTMEvent({ event: 'buttonClicked', value: 'xyz' })}
      >
        Send Event
      </button>
    </div>
  )
}
```

</AppOnly>

<PagesOnly>

```jsx filename="pages/index.js"
import { sendGTMEvent } from '@next/third-parties/google'

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGTMEvent({ event: 'buttonClicked', value: 'xyz' })}
      >
        Send Event
      </button>
    </div>
  )
}
```

</PagesOnly>

Refer to the [Tag Manager](https://developers.google.com/tag-platform/tag-manager/datalayer)
documentation to learn about the different variables and events that can be passed into the
function.

#### Options

Options to pass to the Google Tag Manager. For a full list of options, read the [Google Tag Manager
docs](https://developers.google.com/tag-platform/tag-manager/datalayer).

Name	Type	Description
`gtmId`	Required	Your GTM container id.
`dataLayer`	Optional	Data layer array to instantiate the container with. Defaults to an empty array.
`dataLayerName`	Optional	Name of the data layer. Defaults to `dataLayer`.
`auth`	Optional	Value of authentication parameter (`gtm_auth`) for environment snippets.

| `preview` | Optional | Value of preview parameter (`gtm_preview`) for environment snippets. |

### Google Maps Embed

The `GoogleMapsEmbed` component can be used to add a [Google Maps Embed](https://developers.google.com/maps/documentation/embed/embedding-map) to your page. By
default, it uses the `loading` attribute to lazy-load the embed below the fold.

<AppOnly>

```jsx filename="app/page.js"
import { GoogleMapsEmbed } from '@next/third-parties/google'

export default function Page() {
  return (
    <GoogleMapsEmbed
      apiKey="XYZ"
      height={200}
      width="100%"
      mode="place"
      q="Brooklyn+Bridge,New+York,NY"
    />
  )
}
```

</AppOnly>

<PagesOnly>

```jsx filename="pages/index.js"
import { GoogleMapsEmbed } from '@next/third-parties/google'

export default function Page() {
  return (
    <GoogleMapsEmbed
      apiKey="XYZ"
      height={200}
      width="100%"
      mode="place"
      q="Brooklyn+Bridge,New+York,NY"
    />
  )
}
```

</PagesOnly>

#### Options

Options to pass to the Google Maps Embed. For a full list of options, read the [Google Map Embed docs](https://developers.google.com/maps/documentation/embed/embedding-map).

Name	Type	Description
`apiKey`	Required	Your api key.
`mode`	Required	[Map mode](https://developers.google.com/maps/documentation/embed/embedding-map#choosing_map_modes)
`height`	Optional	Height of the embed. Defaults to `auto`.
`width`	Optional	Width of the embed. Defaults to `auto`.
`style`	Optional	Pass styles to the iframe.
`allowfullscreen`	Optional	Property to allow certain map parts to go full screen.
`loading`	Optional	Defaults to lazy. Consider changing if you know your embed will be above the fold.
`q`	Optional	Defines map marker location. _This may be required depending on the map mode_.
`center`	Optional	Defines the center of the map view.
`zoom`	Optional	Sets initial zoom level of the map.
`maptype`	Optional	Defines type of map tiles to load.
`language`	Optional	Defines the language to use for UI elements and for the display of labels on map tiles.
`region`	Optional	Defines the appropriate borders and labels to display, based on geo-political sensitivities.

### YouTube Embed

The `YouTubeEmbed` component can be used to load and display a YouTube embed. This component loads
faster by using [`lite-youtube-embed`](https://github.com/paulirish/lite-

youtube-embed) under the
hood.

<AppOnly>

```jsx filename="app/page.js"
import { YouTubeEmbed } from '@next/third-parties/google'

export default function Page() {
  return <YouTubeEmbed videoid="ogfYd705cRs" height={400}
params="controls=0" />
}
```

</AppOnly>

<PagesOnly>

```jsx filename="pages/index.js"
import { YouTubeEmbed } from '@next/third-parties/google'

export default function Page() {
  return <YouTubeEmbed videoid="ogfYd705cRs" height={400}
params="controls=0" />
}
```

</PagesOnly>

#### Options

| Name | Type | Description
|
| ----------- | -------- |

--- |
| `videoid` | Required | YouTube video id.
|
| `width` | Optional | Width of the video container. Defaults to `auto`
|
| `height` | Optional | Height of the video container. Defaults to `auto`
|
| `playlabel` | Optional | A visually hidden label for the play button for
accessibility.
|
| `params` | Optional | The video player params defined [here](https://

developers.google.com/youtube/player_parameters#Parameters). <br/> Params are passed as a query param string. <br/> Eg: `params="controls=0&start=10&end=30"` |
| `style` | Optional | Used to apply styles to the video container.
|

---
title: Optimizations
nav_title: Optimizing
description: Optimize your Next.js application for best performance and user experience.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js comes with a variety of built-in optimizations designed to improve your application's speed and [Core Web Vitals](https://web.dev/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 `<img>` 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.

<AppOnly>

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](/docs/app/api-reference/functions/generate-metadata#metadata-object) or a dynamic [`generateMetadata` function](/docs/app/api-reference/functions/generate-metadata#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](/docs/app/api-reference/functions/image-response) constructor.

</AppOnly>

<PagesOnly>

The Head Component in Next.js allows you to modify the `<head>` of a page. Learn more in the [Head Component](/docs/pages/api-reference/components/head) documentation.

</PagesOnly>

## 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 <PagesOnly>[Analytics](/docs/app/building-your-application/optimizing/analytics), </PagesOnly> [OpenTelemetry](/docs/pages/building-your-application/optimizing/open-telemetry)<PagesOnly>,</PagesOnly> and [Instrumentation](/docs/pages/building-your-application/optimizing/instrumentation) guides.

---
title: TypeScript
description: Next.js provides a TypeScript-first development experience for building your React application.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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.

<AppOnly>

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

> **🎥 Watch:** Learn about the built-in TypeScript plugin → [YouTube (3 minutes)](https://www.youtube.com/watch?v=pqMqn9fKEf8)

</AppOnly>

## New Projects

`create-next-app` now ships with TypeScript by default.

```bash filename="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.

If you already had a `jsconfig.json` file, copy the `paths` compiler option from the old `jsconfig.json` into the new `tsconfig.json` file, and delete the old `jsconfig.json` file.

<AppOnly>

## 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.

You can enable the plugin in VS Code by:

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

3. Selecting "Use Workspace Version"

<Image
  alt="TypeScript Command Palette"
  srcLight="/docs/light/typescript-command-palette.png"
  srcDark="/docs/dark/typescript-command-palette.png"
  width="1600"
  height="637"
/>

Now, when editing files, the custom plugin will be enabled. When running `next build`, the custom type checker will be used.

### Plugin Features

The TypeScript plugin can help with:

- Warning if the invalid values for [segment config options](/docs/app/api-reference/file-conventions/route-segment-config) 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.

> **Good to know**: More features will be added in the future.

</AppOnly>

## 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](https://devblogs.microsoft.com/typescript/announcing-typescript-4-5/#type-on-import-names) and [performance improvements](https://devblogs.microsoft.com/typescript/announcing-typescript-4-5/#real-path-sync-native).

<AppOnly>

## 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 filename="next.config.js"
```

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    typedRoutes: true,
  },
}

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`:

```tsx
import type { Route } from 'next';
import Link from 'next/link'

// No TypeScript errors if href is a valid route
<Link href="/about" />
<Link href="/blog/nextjs" />
<Link href={`/blog/${slug}`} />
<Link href={('/blog' + slug) as Route} />

// TypeScript errors if href is not a valid route
<Link href="/aboot" />
```

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

```tsx
import type { Route } from 'next'
import Link from 'next/link'

function Card<T extends string>({ href }: { href: Route<T> | URL }) {
  return (
    <Link href={href}>
      <div>My Card</div>
    </Link>
  )
}
```

> **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. Previously, data flowing between individual `pages` and `_app` were difficult to type and could introduce confusing bugs. With [colocated data fetching](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating) in Next.js 13, this is no longer an issue.

[Data Fetching in Next.js](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating) 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:

```tsx filename="app/page.tsx"
async function getData() {
  const res = await fetch('https://api.example.com/...')
  // The return value is *not* serialized
  // You can return Date, Map, Set, etc.
  return res.json()
}

export default async function Page() {
  const name = await getData()

  return '...'
}
```

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](https://en.wikipedia.org/wiki/Object%E2%80%93relational_mapping) or type-safe query builder.

## Async Server Component TypeScript Error

To use an `async` Server Component with TypeScript, ensure you are using TypeScript `5.1.3` 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.

</AppOnly>

<PagesOnly>

## Static Generation and Server-side Rendering

For [`getStaticProps`](/docs/pages/api-reference/functions/get-static-props), [`getStaticPaths`](/docs/pages/api-reference/functions/get-static-paths), and [`getServerSideProps`](/docs/pages/api-reference/functions/get-server-side-props), you can use the `GetStaticProps`, `GetStaticPaths`, and `GetServerSideProps` types respectively:

```tsx filename="pages/blog/[slug].tsx"
import { GetStaticProps, GetStaticPaths, GetServerSideProps } from 'next'

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

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

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

> **Good to know:** `satisfies` was added to TypeScript in [4.9](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-4-9.html). We recommend upgrading to the latest version of TypeScript.

## API Routes

The following is an example of how to use the built-in types for API routes:

```ts
import type { NextApiRequest, NextApiResponse } from 'next'

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  res.status(200).json({ name: 'John Doe' })
}
```

You can also type the response data:

```ts
import type { NextApiRequest, NextApiResponse } from 'next'

type Data = {
  name: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<Data>
) {
  res.status(200).json({ name: 'John Doe' })
}
```

## Custom `App`

If you have a [custom `App`](/docs/pages/building-your-application/routing/custom-app), you can use the built-in type `AppProps` and change file name to `./pages/_app.tsx` like so:

```ts
import type { AppProps } from 'next/app'

export default function MyApp({ Component, pageProps }: AppProps) {
  return <Component {...pageProps} />
}
```

</PagesOnly>

## Path aliases and baseUrl

Next.js automatically supports the `tsconfig.json` `"paths"` and `"baseUrl"` options.

<AppOnly>

You can learn more about this feature on the [Module Path aliases documentation](/docs/app/building-your-application/configuring/absolute-imports-and-module-aliases).

</AppOnly>

<PagesOnly>

You can learn more about this feature on the [Module Path aliases documentation](/docs/pages/building-your-application/configuring/absolute-imports-and-module-aliases).

</PagesOnly>

## 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:

```js
// @ts-check

/**
 * @type {import('next').NextConfig}
 **/
const nextConfig = {
  /* config options here */
```

}

module.exports = nextConfig

## Incremental type checking

Since `v10.2.1` Next.js supports [incremental type checking](https://www.typescriptlang.org/tsconfig#incremental) 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 filename="next.config.js"
module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

## Version Changes

Version	Changes
`v13.2.0`	Statically typed links are available in beta.
`v12.0.0`	[SWC](/docs/architecture/nextjs-compiler) is now used by default

to compile TypeScript and TSX for faster builds. |
| `v10.2.1` | [Incremental type checking](https://www.typescriptlang.org/tsconfig#incremental) support added when enabled in your `tsconfig.json`. |

---
title: ESLint
description: Next.js provides an integrated ESLint experience by default. These conformance rules help you use Next.js in an optimal way.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js provides an integrated [ESLint](https://eslint.org/) experience out of the box. Add `next lint` as a script to `package.json`:

```json filename="package.json"
{
  "scripts": {
    "lint": "next lint"
  }
}
```

Then run `npm run lint` or `yarn lint`:

```bash filename="Terminal"
yarn lint
```

If you don't already have ESLint configured in your application, you will be guided through the installation and configuration process.

```bash filename="Terminal"
yarn lint
```

> You'll see a prompt like this:
>
> ? How would you like to configure ESLint?
>
> ❯ Strict (recommended)
> Base
> Cancel

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](#core-web-vitals). This is the recommended configuration for developers setting up ESLint for the first time.

  ```json filename=".eslintrc.json"
  {
    "extends": "next/core-web-vitals"
  }
  ```

- **Base**: Includes Next.js' base ESLint configuration.

  ```json filename=".eslintrc.json"
  {
    "extends": "next"
  }
  ```

- **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 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.

<AppOnly>

> If you do not want ESLint to run during `next build`, refer to the documentation for [Ignoring ESLint](/docs/app/api-reference/next-config-js/eslint).

</AppOnly>

<PagesOnly>

> If you do not want ESLint to run during `next build`, refer to the documentation for [Ignoring ESLint](/docs/pages/api-reference/next-config-js/eslint).

</PagesOnly>

We recommend using an appropriate [integration](https://eslint.org/docs/user-guide/integrations#editors) 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](#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`](https://www.npmjs.com/package/eslint-plugin-react)
- [`eslint-plugin-react-hooks`](https://www.npmjs.com/package/eslint-plugin-react-hooks)
- [`eslint-plugin-next`](https://www.npmjs.com/package/@next/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`](https://www.npmjs.com/package/@next/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:

<Check size={18} /> Enabled in the recommended configuration

| | Rule
Description
<Check size={18} />	[@next/next/google-font-display](/docs/messages/google-font-display)	Enforce font-display

behavior with Google Fonts. |
<Check size={18} />	[@next/next/google-font-preconnect](/docs/messages/google-font-preconnect)	Ensure `preconnect` is used with Google Fonts.
<Check size={18} />	[@next/next/inline-script-id](/docs/messages/inline-script-id)	Enforce `id` attribute on `next/script` components with inline content.
<Check size={18} />	[@next/next/next-script-for-ga](/docs/messages/next-script-for-ga)	Prefer `next/script` component when using the inline script for Google Analytics.
<Check size={18} />	[@next/next/no-assign-module-variable](/docs/messages/no-assign-module-variable)	Prevent assignment to the `module` variable.
<Check size={18} />	[@next/next/no-async-client-component](/docs/messages/no-async-client-component)	Prevent client components from being async functions.
<Check size={18} />	[@next/next/no-before-interactive-script-outside-document](/docs/messages/no-before-interactive-script-outside-document)	Prevent usage of `next/script`'s `beforeInteractive` strategy outside of `pages/_document.js`.
<Check size={18} />	[@next/next/no-css-tags](/docs/messages/no-css-tags)	Prevent manual stylesheet tags.
<Check size={18} />	[@next/next/no-document-import-in-page](/docs/messages/no-document-import-in-page)	Prevent importing `next/document` outside of `pages/_document.js`.
<Check size={18} />	[@next/next/no-duplicate-head](/docs/messages/no-duplicate-head)	Prevent duplicate usage of `<Head>` in `pages/_document.js`.
<Check size={18} />	[@next/next/no-head-element](/docs/messages/no-head-element)	Prevent usage of `<head>` element.
<Check size={18} />	[@next/next/no-head-import-in-document](/docs/messages/no-head-import-in-document)	Prevent usage of `next/head` in `pages/_document.js`.
<Check size={18} />	[@next/next/no-html-link-for-pages](/docs/messages/no-html-link-for-pages)	Prevent usage of `<a>` elements to navigate to internal Next.js pages.
<Check size={18} />	[@next/next/no-img-element](/docs/messages/no-img-element)	Prevent usage of `<img>` element due to slower LCP and higher bandwidth.
<Check size={18} />	[@next/next/no-page-custom-font](/docs/messages/no-page-custom-font)	Prevent page-only custom fonts.
<Check size={18} />	[@next/next/no-script-component-in-head](/docs/

messages/no-script-component-in-head) | Prevent usage of `next/script` in `next/head` component. |
| <Check size={18} /> | [@next/next/no-styled-jsx-in-document](/docs/messages/no-styled-jsx-in-document) | Prevent usage of `styled-jsx` in `pages/_document.js`. |
| <Check size={18} /> | [@next/next/no-sync-scripts](/docs/messages/no-sync-scripts) | Prevent synchronous scripts.
|
| <Check size={18} /> | [@next/next/no-title-in-document-head](/docs/messages/no-title-in-document-head) | Prevent usage of `<title>` with `Head` component from `next/document`.
|
| <Check size={18} /> | @next/next/no-typos
| Prevent common typos in [Next.js's data fetching functions](/docs/pages/building-your-application/data-fetching) |
| <Check size={18} /> | [@next/next/no-unwanted-polyfillio](/docs/messages/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](#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`:

```json filename=".eslintrc.json"
{
  "extends": "next",
  "settings": {
    "next": {
      "rootDir": "packages/my-app/"
    }
  }
}
```

`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/`, `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 filename="next.config.js"
module.exports = {
  eslint: {
    dirs: ['pages', 'utils'], // Only run ESLint on the 'pages' and 'utils' directories during production builds (next build)
  },
}
```

Similarly, the `--dir` and `--file` flags can be used for `next lint` to lint specific directories and files:

```bash filename="Terminal"
next lint --dir pages --dir utils --file bar.js
```

## Caching

<AppOnly>

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](/docs/app/api-reference/next-config-js/distDir). 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`.

</AppOnly>

<PagesOnly>

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](/docs/pages/api-reference/next-config-js/distDir). 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`.

</PagesOnly>

```bash filename="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`:

```json filename=".eslintrc.json"
{
  "extends": "next",
  "rules": {
    "react/no-unescaped-entities": "off",
    "@next/next/no-page-custom-font": "off"
  }
}
```

### 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.

```json filename=".eslintrc.json"
{
  "extends": "next/core-web-vitals"
}
```

`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](https://web.dev/vitals/).

> The `next/core-web-vitals` entry point is automatically included for new applications built with [Create Next App](/docs/app/api-reference/create-next-app).

## Usage With Other Tools

### Prettier

ESLint also contains code formatting rules, which can conflict with your existing [Prettier](https://prettier.io/) setup. We recommend including [eslint-config-prettier](https://github.com/prettier/eslint-config-prettier) in your ESLint config to make ESLint and Prettier work together.

First, install the dependency:

```bash filename="Terminal"
npm install --save-dev eslint-config-prettier

yarn add --dev eslint-config-prettier

pnpm add --save-dev eslint-config-prettier

bun add --dev eslint-config-prettier
```

Then, add `prettier` to your existing ESLint config:

```json filename=".eslintrc.json"
{
  "extends": ["next", "prettier"]
}
```

### lint-staged

If you would like to use `next lint` with [lint-staged](https://github.com/okonet/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 filename=".lintstagedrc.js"
const path = require('path')

const buildEslintCommand = (filenames) =>
  `next lint --fix --file ${filenames
    .map((f) => path.relative(process.cwd(), f))
    .join(' --file ')}`

module.exports = {
  '*.{js,jsx,ts,tsx}': [buildEslintCommand],
}
```

## 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](/docs/pages/building-your-application/configuring/babel))
- You have `eslint-plugin-import` installed with Node.js and/or TypeScript [resolvers](https://github.com/benmosher/eslint-plugin-import#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`](https://github.com/vercel/next.js/blob/canary/packages/eslint-config-next/index.js) or extending directly from the Next.js ESLint plugin instead:

```js
module.exports = {
  extends: [
    //...
    'plugin:@next/next/recommended',
  ],
}
```

The plugin can be installed normally in your project without needing to run `next lint`:

```bash filename="Terminal"
npm install --save-dev @next/eslint-plugin-next

yarn add --dev @next/eslint-plugin-next

pnpm add --save-dev @next/eslint-plugin-next

bun 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:

```json filename=".eslintrc.json"
{
  "extends": ["eslint:recommended", "next"]
}
```

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.

---
title: Environment Variables
description: Learn to add and access environment variables in your Next.js application.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<details>
  <summary>Examples</summary>

- [Environment Variables](https://github.com/vercel/next.js/tree/canary/examples/environment-variables)

</details>

Next.js comes with built-in support for environment variables, which allows you to do the following:

- [Use `.env.local` to load environment variables](#loading-environment-variables)
- [Bundle environment variables for the browser by prefixing with

`NEXT_PUBLIC_`](#bundling-environment-variables-for-the-browser)

## Loading Environment Variables

Next.js has built-in support for loading environment variables from `.env.local` into `process.env`.

```txt filename=".env.local"
DB_HOST=localhost
DB_USER=myuser
DB_PASS=mypassword
```

<PagesOnly>

This loads `process.env.DB_HOST`, `process.env.DB_USER`, and `process.env.DB_PASS` into the Node.js environment automatically allowing you to use them in [Next.js data fetching methods](/docs/pages/building-your-application/data-fetching) and [API routes](/docs/pages/building-your-application/routing/api-routes).

For example, using [`getStaticProps`](/docs/pages/building-your-application/data-fetching/get-static-props):

```js filename="pages/index.js"
export async function getStaticProps() {
  const db = await myDB.connect({
    host: process.env.DB_HOST,
    username: process.env.DB_USER,
    password: process.env.DB_PASS,
  })
  // ...
}
```

</PagesOnly>

<AppOnly>

> **Note**: Next.js also supports multiline variables inside of your `.env*` files:
>
> ```bash
> # .env.local
>
> # you can write with line breaks
> PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
> ...

> Kh9NV...
> ...
> -----END DSA PRIVATE KEY-----"
>
> # or with `\n` inside double quotes
> PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----\nKh9NV...\n-----END DSA PRIVATE KEY-----\n"
> ```

> **Note**: If you are using a `/src` folder, please note that Next.js will load the .env files **only** from the parent folder and **not** from the `/src` folder.
> 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](/docs/app/building-your-application/routing/route-handlers).

For example:

```js filename="app/api/route.js"
export async function GET() {
  const db = await myDB.connect({
    host: process.env.DB_HOST,
    username: process.env.DB_USER,
    password: process.env.DB_PASS,
  })
  // ...
}
```

</AppOnly>

### 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:

```txt filename=".env"
TWITTER_USER=nextjs
TWITTER_URL=https://twitter.com/$TWITTER_USER
```

In the above example, `process.env.TWITTER_URL` would be set to `https://twitter.com/nextjs`.

> **Good to know**: If you need to use variable with a `$` in the actual value, it needs to be escaped e.g. `\$`.

## Bundling Environment Variables for the Browser

Non-`NEXT_PUBLIC_` environment variables are only available in the Node.js environment, meaning they aren't accessible to the browser (the client runs in a different _environment_).

In order to make the value of an environment variable accessible in the browser, Next.js can "inline" a value, at build time, into the js bundle that is delivered to the client, replacing all references to `process.env.[variable]` with a hard-coded value. To tell it to do this, you just have to prefix the variable with `NEXT_PUBLIC_`. For example:

```txt filename="Terminal"
NEXT_PUBLIC_ANALYTICS_ID=abcdefghijk
```

This will tell Next.js to replace all references to `process.env.NEXT_PUBLIC_ANALYTICS_ID` in the Node.js environment with the value from the environment in which you run `next build`, allowing you to use it anywhere in your code. It will be inlined into any JavaScript sent to the browser.

> **Note**: After being built, your app will no longer respond to changes to these environment variables. For instance, if you use a Heroku pipeline to promote slugs built in one environment to another environment, or if you build and deploy a single Docker image to multiple environments, all `NEXT_PUBLIC_` variables will be frozen with the value evaluated at build time, so these values need to be set appropriately when the project is built. If you need access to runtime environment values, you'll have to setup your own API to provide them to the client (either on demand or during initialization).

```js filename="pages/index.js"
import setupAnalyticsService from '../lib/my-analytics-service'

// 'NEXT_PUBLIC_ANALYTICS_ID' can be used here as it's prefixed by 'NEXT_PUBLIC_'.
// It will be transformed at build time to `setupAnalyticsService('abcdefghijk')`.
setupAnalyticsService(process.env.NEXT_PUBLIC_ANALYTICS_ID)

function HomePage() {
  return <h1>Hello World</h1>
}

export default HomePage
```

Note that dynamic lookups will _not_ be inlined, such as:

```js
// This will NOT be inlined, because it uses a variable
const varName = 'NEXT_PUBLIC_ANALYTICS_ID'
setupAnalyticsService(process.env[varName])

// This will NOT be inlined, because it uses a variable
const env = process.env
setupAnalyticsService(env.NEXT_PUBLIC_ANALYTICS_ID)
```

### Runtime Environment Variables

Next.js can support both build time and runtime environment variables.

By default, environment variables are only available on the server. To expose an environment variable to the browser, it must be prefixed with `NEXT_PUBLIC_`. However, these public environment variables will be inlined into the JavaScript bundle during `next build`.

To read runtime environment variables, we recommend using `getServerSideProps` or [incrementally adopting the App Router](/docs/app/building-your-application/upgrading/app-router-migration). With the App Router, we can safely read environment variables on the server during dynamic rendering. This allows you to use a singular Docker image that can be promoted through multiple environments with different values.

```jsx
import { unstable_noStore as noStore } from 'next/cache'

export default function Component() {
  noStore()
  // cookies(), headers(), and other dynamic functions
  // will also opt into dynamic rendering, making
  // this env variable is evaluated at runtime
  const value = process.env.MY_VALUE
  // ...
}
```

Good to know:

- You can run code on server startup using the [`register` function](/docs/app/building-your-application/optimizing/instrumentation).
- We do not recommend using the [runtimeConfig](/docs/pages/api-reference/

next-config-js/runtime-configuration) option, as this does not work with the standalone output mode. Instead, we recommend [incrementally adopting](/docs/app/building-your-application/upgrading/app-router-migration) the App Router.

## 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.

> **Good to know**: `.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](https://vercel.com), Environment Variables can be configured [in the Project Settings](https://vercel.com/docs/concepts/projects/environment-variables?utm_source=next-site&utm_medium=docs&utm_campaign=next-website).

All types of Environment Variables should be configured there. Even Environment Variables used in Development – which can be [downloaded onto your local device](https://vercel.com/docs/concepts/projects/environment-variables#development-environment-variables?utm_source=next-site&utm_medium=docs&utm_campaign=next-website) afterwards.

If you've configured [Development Environment Variables](https://vercel.com/docs/concepts/projects/environment-variables#development-environment-variables?utm_source=next-site&utm_medium=docs&utm_campaign=next-website) you can pull them into a `.env.local` for usage on your local machine using the following command:

```bash filename="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).

> **Good to know**: 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.

```js
// The below can be used in a Jest global setup file or similar for your testing set-up
import { loadEnvConfig } from '@next/env'

export default async () => {
  const projectDir = process.cwd()
  loadEnvConfig(projectDir)
}
```

## Environment Variable Load Order

Environment variables are looked up in the following places, in order, stopping once the variable is found.

1. `process.env`
1. `.env.$(NODE_ENV).local`
1. `.env.local` (Not checked when `NODE_ENV` is `test`.)

1. `.env.$(NODE_ENV)`
1. `.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.

> **Good to know**: The allowed values for `NODE_ENV` are `production`, `development` and `test`.

## Good to know

- If you are using a [`/src` directory](/docs/app/building-your-application/configuring/src-directory), `.env.*` files should remain in the root of your project.
- If the environment variable `NODE_ENV` is unassigned, Next.js automatically assigns `development` when running the `next dev` command, or `production` for all other commands.

## Version History

Version	Changes
`v9.4.0`	Support `.env` and `NEXT_PUBLIC_` introduced.

---
title: Absolute Imports and Module Path Aliases
description: Configure module path aliases that allow you to remap certain import paths.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<details>
  <summary>Examples</summary>

- [Absolute Imports and Aliases](https://github.com/vercel/next.js/tree/canary/examples/with-absolute-imports)

</details>

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:

```tsx
// before
import { Button } from '../../../components/button'

// after
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:

```json filename="tsconfig.json or jsconfig.json"
{
  "compilerOptions": {
    "baseUrl": "."
  }
}
```

```tsx filename="components/button.tsx" switcher
export default function Button() {
  return <button>Click me</button>
}
```

```jsx filename="components/button.js" switcher
export default function Button() {
  return <button>Click me</button>
}
```

```tsx filename="app/page.tsx" switcher
import Button from 'components/button'

export default function HomePage() {
  return (
```

    <>
      <h1>Hello World</h1>
      <Button />
    </>
)
}

```jsx filename="app/page.js" switcher
import Button from 'components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
  )
}
```

## 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/*`:

```json filename="tsconfig.json or jsconfig.json"
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/components/*": ["components/*"]
    }
  }
}
```

```tsx filename="components/button.tsx" switcher
export default function Button() {
  return <button>Click me</button>
}
```

```jsx filename="components/button.js" switcher
```

~~~
export default function Button() {
  return <button>Click me</button>
}
```

```tsx filename="app/page.tsx" switcher
import Button from '@/components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
  )
}
```

```jsx filename="app/page.js" switcher
import Button from '@/components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
  )
}
```

Each of the `"paths"` are relative to the `baseUrl` location. For example:

```json
// tsconfig.json or jsconfig.json
{
  "compilerOptions": {
    "baseUrl": "src/",
    "paths": {
      "@/styles/*": ["styles/*"],
      "@/components/*": ["components/*"]
    }
  }
}
```

```jsx
~~~

// pages/index.js
import Button from '@/components/button'
import '@/styles/styles.css'
import Helper from 'utils/helper'

export default function HomePage() {
  return (
    <Helper>
      <h1>Hello World</h1>
      <Button />
    </Helper>
)
}

---
title: Markdown and MDX
nav_title: MDX
description: Learn how to configure MDX to write JSX in your markdown files.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

[Markdown](https://daringfireball.net/projects/markdown/syntax) 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...

```md
I **love** using [Next.js](https://nextjs.org/)
```

Output:

```html
<p>I <strong>love</strong> using <a href="https://nextjs.org/">Next.js</a></p>
```

[MDX](https://mdxjs.com/) is a superset of markdown that lets you write [JSX](https://react.dev/learn/writing-markup-with-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 (the default in App Router).

## `@next/mdx`

The `@next/mdx` package is used to configure Next.js so it can process markdown and MDX. **It sources data from local files**, allowing you to create pages with a `.mdx` extension, directly in your `/pages` or `/app` directory.

Let's walk through how to configure and use MDX with Next.js.

## Getting Started

Install packages needed to render MDX:

```bash filename="Terminal"
npm install @next/mdx @mdx-js/loader @mdx-js/react @types/mdx
```

<AppOnly>

Create a `mdx-components.tsx` file at the root of your application (the parent folder of `app/` or `src/`):

> **Good to know**: `mdx-components.tsx` is required to use MDX with App Router and will not work without it.

```tsx filename="mdx-components.tsx" switcher
import type { MDXComponents } from 'mdx/types'

export function useMDXComponents(components: MDXComponents):
MDXComponents {
  return {
    ...components,
  }
}
```

```js filename="mdx-components.js" switcher
export function useMDXComponents(components) {
  return {
    ...components,
```

  }
}

</AppOnly>

Update the `next.config.js` file at your project's root to configure it to use MDX:

```js filename="next.config.js"
const withMDX = require('@next/mdx')()

/** @type {import('next').NextConfig} */
const nextConfig = {
  // Configure `pageExtensions` to include MDX files
  pageExtensions: ['js', 'jsx', 'mdx', 'ts', 'tsx'],
  // Optionally, add any other Next.js config below
}

module.exports = withMDX(nextConfig)
```

<AppOnly>

Then, create a new MDX page within the `/app` directory:

```txt
  your-project
  ├── app
  │   └── my-mdx-page
  │       └── page.mdx
  └── package.json
```

</AppOnly>

<PagesOnly>

Then, create a new MDX page within the `/pages` directory:

```txt
  your-project
  ├── pages
  │   └── my-mdx-page.mdx
  └── package.json
```

</PagesOnly>

Now you can use markdown and import React components directly inside your MDX page:

```mdx
import { MyComponent } from 'my-components'

# Welcome to my MDX page!

This is some **bold** and _italics_ text.

This is a list in markdown:

- One
- Two
- Three

Checkout my React component:

<MyComponent />
```

Navigating to the `/my-mdx-page` route should display your rendered MDX.

## Remote MDX

If your markdown or MDX files or content lives _somewhere else_, you can fetch it dynamically on the server. This is useful for content stored in a separate local folder, CMS, database, or anywhere else.

There are two popular community packages for fetching MDX content:

- [`next-mdx-remote`](https://github.com/hashicorp/next-mdx-remote#react-server-components-rsc--nextjs-app-directory-support)
- [`contentlayer`](https://www.contentlayer.dev/)

> **Good to know**: 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).

The following example uses `next-mdx-remote`:

<AppOnly>

```tsx filename="app/my-mdx-page-remote/page.tsx" switcher
```

~~~
import { MDXRemote } from 'next-mdx-remote/rsc'

export default async function RemoteMdxPage() {
  // MDX text - can be from a local file, database, CMS, fetch, anywhere...
  const res = await fetch('https://...')
  const markdown = await res.text()
  return <MDXRemote source={markdown} />
}
```

```jsx filename="app/my-mdx-page-remote/page.js" switcher
import { MDXRemote } from 'next-mdx-remote/rsc'

export default async function RemoteMdxPage() {
  // MDX text - can be from a local file, database, CMS, fetch, anywhere...
  const res = await fetch('https://...')
  const markdown = await res.text()
  return <MDXRemote source={markdown} />
}
```

</AppOnly>

<PagesOnly>

```tsx filename="pages/my-mdx-page-remote.tsx" switcher
import { serialize } from 'next-mdx-remote/serialize'
import { MDXRemote, MDXRemoteSerializeResult } from 'next-mdx-remote'

interface Props {
  mdxSource: MDXRemoteSerializeResult
}

export default function RemoteMdxPage({ mdxSource }: Props) {
  return <MDXRemote {...mdxSource} />
}

export async function getStaticProps() {
  // MDX text - can be from a local file, database, CMS, fetch, anywhere...
  const res = await fetch('https:...')
  const mdxText = await res.text()
  const mdxSource = await serialize(mdxText)
  return { props: { mdxSource } }
}
```

```jsx filename="pages/my-mdx-page-remote.js" switcher
~~~

```
import { serialize } from 'next-mdx-remote/serialize'
import { MDXRemote } from 'next-mdx-remote'

export default function RemoteMdxPage({ mdxSource }) {
  return <MDXRemote {...mdxSource} />
}

export async function getStaticProps() {
  // MDX text - can be from a local file, database, CMS, fetch, anywhere...
  const res = await fetch('https:...')
  const mdxText = await res.text()
  const mdxSource = await serialize(mdxText)
  return { props: { mdxSource } }
}
```

</PagesOnly>

Navigating to the `/my-mdx-page-remote` route should display your rendered MDX.

## Layouts

<AppOnly>

To share a layout amongst MDX pages, you can use the [built-in layouts support](/docs/app/building-your-application/routing/pages-and-layouts#layouts) with the App Router.

```tsx filename="app/my-mdx-page/layout.tsx" switcher
export default function MdxLayout({ children }: { children: React.ReactNode }) {
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}</div>
}
```

```jsx filename="app/my-mdx-page/layout.js" switcher
export default function MdxLayout({ children }) {
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}</div>
}
```

</AppOnly>

<PagesOnly>

To share a layout around MDX pages, create a layout component:

```tsx filename="components/mdx-layout.tsx" switcher
export default function MdxLayout({ children }: { children: React.ReactNode }) {
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}</div>
}
```

```jsx filename="components/mdx-layout.js" switcher
export default function MdxLayout({ children }) {
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}</div>
}
```

Then, import the layout component into the MDX page, wrap the MDX content in the layout, and export it:

```mdx
import MdxLayout from '../components/mdx-layout'

# Welcome to my MDX page!

export default function MDXPage({ children }) {
  return <MdxLayout>{children}</MdxLayout>;

}
```

</PagesOnly>

## 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 filename="next.config.mjs"
import remarkGfm from 'remark-gfm'
import createMDX from '@next/mdx'
```

/** @type {import('next').NextConfig} */
const nextConfig = {
  // Configure `pageExtensions`` to include MDX files
  pageExtensions: ['js', 'jsx', 'mdx', 'ts', 'tsx'],
  // Optionally, add any other Next.js config below
}

const withMDX = createMDX({
  // Add markdown plugins here, as desired
  options: {
    remarkPlugins: [remarkGfm],
    rehypePlugins: [],
  },
})

// Merge MDX config with Next.js config
export default 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:

- [remark-frontmatter](https://github.com/remarkjs/remark-frontmatter)
- [gray-matter](https://github.com/jonschlinkert/gray-matter).

To access page metadata with `@next/mdx`, you can export a metadata object from within the `.mdx` file:

```mdx
export const metadata = {
  author: 'John Doe',
}

# 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:

```md
```

This is a list in markdown:

- One
- Two
- Three

The above generates the following `HTML`:

```html
<p>This is a list in markdown:</p>

<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
</ul>
```

When you want to style your own elements for a custom feel to your website or application, you can pass in shortcodes. These are your own custom components that map to `HTML` elements.

<AppOnly>

To do this, open the `mdx-components.tsx` file at the root of your application and add custom elements:

</AppOnly>

<PagesOnly>

To do this, create a `mdx-components.tsx` file at the root of your application (the parent folder of `pages/` or `src/`) and add custom elements:

</PagesOnly>

```tsx filename="mdx-components.tsx" switcher
import type { MDXComponents } from 'mdx/types'
import Image from 'next/image'

// This file allows you to provide custom React components
// to be used in MDX files. You can import and use any
// React component you want, including inline styles,
// components from other libraries, and more.

export function useMDXComponents(components: MDXComponents):
```

```
MDXComponents {
  return {
    // Allows customizing built-in components, e.g. to add styling.
    h1: ({ children }) => <h1 style={{ fontSize: '100px' }}>{children}</h1>,
    img: (props) => (
      <Image
        sizes="100vw"
        style={{ width: '100%', height: 'auto' }}
        {...props}
      />
    ),
    ...components,
  }
}
```

```js filename="mdx-components.js" switcher
import Image from 'next/image'

// This file allows you to provide custom React components
// to be used in MDX files. You can import and use any
// React component you want, including inline styles,
// components from other libraries, and more.

export function useMDXComponents(components) {
  return {
    // Allows customizing built-in components, e.g. to add styling.
    h1: ({ children }) => <h1 style={{ fontSize: '100px' }}>{children}</h1>,
    img: (props) => (
      <Image
        sizes="100vw"
        style={{ width: '100%', height: 'auto' }}
        {...props}
      />
    ),
    ...components,
  }
}
```

## 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:

```js
import { unified } from 'unified'
import remarkParse from 'remark-parse'
import remarkRehype from 'remark-rehype'
import rehypeSanitize from 'rehype-sanitize'
import rehypeStringify from 'rehype-stringify'

main()

async function main() {
  const file = await unified()
    .use(remarkParse) // Convert into markdown AST
    .use(remarkRehype) // Transform to HTML AST
    .use(rehypeSanitize) // Sanitize HTML input
    .use(rehypeStringify) // Convert AST into serialized HTML
    .process('Hello, Next.js!')

  console.log(String(file)) // <p>Hello, Next.js!</p>
}
```

The `remark` and `rehype` ecosystem contains plugins for [syntax highlighting](https://github.com/atomiks/rehype-pretty-code), [linking headings](https://github.com/rehypejs/rehype-autolink-headings), [generating a table of contents](https://github.com/remarkjs/remark-toc), and more.

When using `@next/mdx` as shown above, you **do not** need to use `remark` or `rehype` directly, as it is handled for you. We're describing it here for a deeper understanding of what the `@next/mdx` package is doing underneath.

## 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 filename="next.config.js"
module.exports = withMDX({
  experimental: {
    mdxRs: true,
  },
})
```

```

## Helpful Links

- [MDX](https://mdxjs.com)
- [`@next/mdx`](https://www.npmjs.com/package/@next/mdx)
- [remark](https://github.com/remarkjs/remark)
- [rehype](https://github.com/rehypejs/rehype)

---
title: src Directory
description: Save pages under the `src` directory as an alternative to the root `pages` directory.
related:
  links:
    - app/building-your-application/routing/colocation
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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.

<Image
  alt="An example folder structure with the `src` directory"
  srcLight="/docs/light/project-organization-src-directory.png"
  srcDark="/docs/dark/project-organization-src-directory.png"
  width="1600"
  height="687"
/>

> **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`.
> - If you're using Middleware, ensure it is placed inside the `src` directory.
> - If you're using Tailwind CSS, you'll need to add the `/src` prefix to the `tailwind.config.js` file in the [content section](https://tailwindcss.com/docs/content-configuration).

---
title: Draft Mode
description: Next.js has draft mode to toggle between static and dynamic pages. You can learn how it works with App Router here.

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](/docs/app/building-your-application/rendering/server-components#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](/docs/app/building-your-application/routing/route-handlers). It can have any name - e.g. `app/api/draft/route.ts`

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

```ts filename="app/api/draft/route.ts" switcher
// route handler enabling draft mode
import { draftMode } from 'next/headers'

export async function GET(request: Request) {
  draftMode().enable()
  return new Response('Draft mode is enabled')
}
```

```js filename="app/api/draft/route.js" switcher
// route handler enabling draft mode
import { draftMode } from 'next/headers'
```

export async function GET(request) {
  draftMode().enable()
  return new Response('Draft mode is enabled')
}

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`

```bash filename="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 filename="app/api/draft/route.ts" switcher
// route handler with secret and slug
import { draftMode } from 'next/headers'
import { redirect } from 'next/navigation'

export async function GET(request: Request) {
  // Parse query string parameters
  const { searchParams } = new URL(request.url)
  const secret = searchParams.get('secret')
  const slug = searchParams.get('slug')

  // Check the secret and next parameters
  // This secret should only be known to this route handler and the CMS
  if (secret !== 'MY_SECRET_TOKEN' || !slug) {
    return new Response('Invalid token', { status: 401 })
  }

  // Fetch the headless CMS to check if the provided `slug` exists
  // getPostBySlug would implement the required fetching logic to the headless CMS
  const post = await getPostBySlug(slug)

  // If the slug doesn't exist prevent draft mode from being enabled
  if (!post) {
    return new Response('Invalid slug', { status: 401 })
  }

  // Enable Draft Mode by setting the cookie
  draftMode().enable()

  // Redirect to the path from the fetched post
  // We don't redirect to searchParams.slug as that might lead to open redirect vulnerabilities
  redirect(post.slug)
}
```

```js filename="app/api/draft/route.js" switcher
// route handler with secret and slug
```

```
import { draftMode } from 'next/headers'
import { redirect } from 'next/navigation'

export async function GET(request) {
  // Parse query string parameters
  const { searchParams } = new URL(request.url)
  const secret = searchParams.get('secret')
  const slug = searchParams.get('slug')

  // Check the secret and next parameters
  // This secret should only be known to this route handler and the CMS
  if (secret !== 'MY_SECRET_TOKEN' || !slug) {
    return new Response('Invalid token', { status: 401 })
  }

  // Fetch the headless CMS to check if the provided `slug` exists
  // getPostBySlug would implement the required fetching logic to the headless
CMS
  const post = await getPostBySlug(slug)

  // If the slug doesn't exist prevent draft mode from being enabled
  if (!post) {
    return new Response('Invalid slug', { status: 401 })
  }

  // Enable Draft Mode by setting the cookie
  draftMode().enable()

  // Redirect to the path from the fetched post
  // We don't redirect to searchParams.slug as that might lead to open redirect
vulnerabilities
  redirect(post.slug)
}
```

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`.

```tsx filename="app/page.tsx" switcher
// page that fetches data
import { draftMode } from 'next/headers'

async function getData() {
  const { isEnabled } = draftMode()

  const url = isEnabled
    ? 'https://draft.example.com'
    : 'https://production.example.com'

  const res = await fetch(url)

  return res.json()
}

export default async function Page() {
  const { title, desc } = await getData()

  return (
    <main>
      <h1>{title}</h1>
      <p>{desc}</p>
    </main>
  )
}
```

```jsx filename="app/page.js" switcher
// page that fetches data
import { draftMode } from 'next/headers'

async function getData() {
  const { isEnabled } = draftMode()

  const url = isEnabled
    ? 'https://draft.example.com'
    : 'https://production.example.com'

  const res = await fetch(url)

  return res.json()
}

export default async function Page() {
  const { title, desc } = await getData()
```

  return (
    <main>
      <h1>{title}</h1>
      <p>{desc}</p>
    </main>
)
}

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.

```bash filename="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 filename="app/api/disable-draft/route.ts" switcher
import { draftMode } from 'next/headers'

export async function GET(request: Request) {
  draftMode().disable()
  return new Response('Draft mode is disabled')
}
```

```js filename="app/api/disable-draft/route.js" switcher
import { draftMode } from 'next/headers'

export async function GET(request) {
  draftMode().disable()
  return new Response('Draft mode is disabled')
}
```

```

Then, send a request to `/api/disable-draft` to invoke the Route Handler. If calling this route using [`next/link`](/docs/app/api-reference/components/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.

> **Good to know**: To test Draft Mode locally over HTTP, your browser will need to allow third-party cookies and local storage access.

---
title: Content Security Policy
description: Learn how to set a Content Security Policy (CSP) for your Next.js application.
related:
  links:
    - app/building-your-application/routing/middleware
    - app/api-reference/functions/headers
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

[Content Security Policy (CSP)](https://developer.mozilla.org/docs/Web/HTTP/CSP) is important to guard your Next.js application against various security threats such as cross-site scripting (XSS), clickjacking, and other code injection attacks.

By using CSP, developers can specify which origins are permissible for content sources, scripts, stylesheets, images, fonts, objects, media (audio, video), iframes, and more.

<details>
  <summary>Examples</summary>

- [Strict CSP](https://github.com/vercel/next.js/tree/canary/examples/with-strict-csp)

</details>
```

## Nonces

A [nonce](https://developer.mozilla.org/docs/Web/HTML/Global_attributes/nonce) is a unique, random string of characters created for a one-time use. It is used in conjunction with CSP to selectively allow certain inline scripts or styles to execute, bypassing strict CSP directives.

### Why use a nonce?

Even though CSPs are designed to block malicious scripts, there are legitimate scenarios where inline scripts are necessary. In such cases, nonces offer a way to allow these scripts to execute if they have the correct nonce.

### Adding a nonce with Middleware

[Middleware](/docs/app/building-your-application/routing/middleware) enables you to add headers and generate nonces before the page renders.

Every time a page is viewed, a fresh nonce should be generated. This means that you **must use dynamic rendering to add nonces**.

For example:

```ts filename="middleware.ts" switcher
import { NextRequest, NextResponse } from 'next/server'

export function middleware(request: NextRequest) {
  const nonce = Buffer.from(crypto.randomUUID()).toString('base64')
  const cspHeader = `
    default-src 'self';
    script-src 'self' 'nonce-${nonce}' 'strict-dynamic';
    style-src 'self' 'nonce-${nonce}';
    img-src 'self' blob: data:;
    font-src 'self';
    object-src 'none';
    base-uri 'self';
    form-action 'self';
    frame-ancestors 'none';
    block-all-mixed-content;
    upgrade-insecure-requests;
`
  // Replace newline characters and spaces
  const contentSecurityPolicyHeaderValue = cspHeader
    .replace(/|s{2,}/g, ' ')
    .trim()
```

~~~
  const requestHeaders = new Headers(request.headers)
  requestHeaders.set('x-nonce', nonce)

  requestHeaders.set(
    'Content-Security-Policy',
    contentSecurityPolicyHeaderValue
  )

  const response = NextResponse.next({
    request: {
      headers: requestHeaders,
    },
  })
  response.headers.set(
    'Content-Security-Policy',
    contentSecurityPolicyHeaderValue
  )

  return response
}
```

```js filename="middleware.js" switcher
import { NextResponse } from 'next/server'

export function middleware(request) {
  const nonce = Buffer.from(crypto.randomUUID()).toString('base64')
  const cspHeader = `
    default-src 'self';
    script-src 'self' 'nonce-${nonce}' 'strict-dynamic';
    style-src 'self' 'nonce-${nonce}';
    img-src 'self' blob: data:;
    font-src 'self';
    object-src 'none';
    base-uri 'self';
    form-action 'self';
    frame-ancestors 'none';
    block-all-mixed-content;
    upgrade-insecure-requests;
`
  // Replace newline characters and spaces
  const contentSecurityPolicyHeaderValue = cspHeader
    .replace(/|s{2,}/g, ' ')
    .trim()

  const requestHeaders = new Headers(request.headers)
~~~

```
  requestHeaders.set('x-nonce', nonce)
  requestHeaders.set(
    'Content-Security-Policy',
    contentSecurityPolicyHeaderValue
  )

  const response = NextResponse.next({
    request: {
      headers: requestHeaders,
    },
  })
  response.headers.set(
    'Content-Security-Policy',
    contentSecurityPolicyHeaderValue
  )

  return response
}
```

By default, Middleware runs on all requests. You can filter Middleware to run on specific paths using a [`matcher`](/docs/app/building-your-application/routing/middleware#matcher).

We recommend ignoring matching prefetches (from `next/link`) and static assets that don't need the CSP header.

```ts filename="middleware.ts" switcher
export const config = {
  matcher: [
    /*
     * Match all request paths except for the ones starting with:
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization files)
     * - favicon.ico (favicon file)
     */
    {
      source: '/((?!api|_next/static|_next/image|favicon.ico).*)',
      missing: [
        { type: 'header', key: 'next-router-prefetch' },
        { type: 'header', key: 'purpose', value: 'prefetch' },
      ],
    },
  ],
}
```

```js filename="middleware.js" switcher
export const config = {
  matcher: [
    /*
     * Match all request paths except for the ones starting with:
     * - api (API routes)
     * - _next/static (static files)
     * - _next/image (image optimization files)
     * - favicon.ico (favicon file)
     */
    {
      source: '/((?!api|_next/static|_next/image|favicon.ico).*)',
      missing: [
        { type: 'header', key: 'next-router-prefetch' },
        { type: 'header', key: 'purpose', value: 'prefetch' },
      ],
    },
  ],
}
```

### Reading the nonce

You can now read the nonce from a [Server Component](/docs/app/building-your-application/rendering/server-components) using [`headers`](/docs/app/api-reference/functions/headers):

```tsx filename="app/page.tsx" switcher
import { headers } from 'next/headers'
import Script from 'next/script'

export default function Page() {
  const nonce = headers().get('x-nonce')

  return (
    <Script
      src="https://www.googletagmanager.com/gtag/js"
      strategy="afterInteractive"
      nonce={nonce}
    />
  )
}
```

```jsx filename="app/page.jsx" switcher
import { headers } from 'next/headers'
import Script from 'next/script'

export default function Page() {
  const nonce = headers().get('x-nonce')

  return (
    <Script
      src="https://www.googletagmanager.com/gtag/js"
      strategy="afterInteractive"
      nonce={nonce}
    />
  )
}
```

## Version History

We recommend using `v13.4.20+` of Next.js to properly handle and apply nonces.

---
title: Configuring
description: Learn how to configure your Next.js application.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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.

---
title: Static Exports
description: Next.js enables starting as a static site or Single-Page Application (SPA), then later optionally upgrading to use features that require a server.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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.

<PagesOnly>

> **Good to know**: We recommend using the App Router for enhanced static export support.

</PagesOnly>

## Configuration

To enable a static export, change the output mode inside `next.config.js`:

```js filename="next.config.js" highlight={5}
/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  output: 'export',

  // Optional: Change links `/me` -> `/me/` and emit `/me.html` -> `/me/index.html`
  // trailingSlash: true,

  // Optional: Prevent automatic `/me` -> `/me/`, instead preserve `href`
  // skipTrailingSlashRedirect: true,

  // Optional: Change the output directory `out` -> `dist`
  // distDir: 'dist',
}

module.exports = nextConfig
```

After running `next build`, Next.js will produce an `out` folder which contains the HTML/CSS/JS assets for your application.

<PagesOnly>

You can utilize [`getStaticProps`](/docs/pages/building-your-application/data-fetching/get-static-props) and [`getStaticPaths`](/docs/pages/building-your-application/data-fetching/get-static-paths) to generate an HTML file for each page in your `pages` directory (or more for [dynamic routes](/docs/app/building-your-application/routing/dynamic-routes)).

</PagesOnly>

<AppOnly>

## 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](#unsupported-features).

```tsx filename="app/page.tsx" switcher
export default async function Page() {
  // This fetch will run on the server during `next build`
  const res = await fetch('https://api.example.com/...')
  const data = await res.json()

  return <main>...</main>
}
```

### Client Components

If you want to perform data fetching on the client, you can use a Client Component with [SWR](https://github.com/vercel/swr) to memoize requests.

```tsx filename="app/other/page.tsx" switcher
'use client'

import useSWR from 'swr'

const fetcher = (url: string) => fetch(url).then((r) => r.json())

export default function Page() {
```

~~~
  const { data, error } = useSWR(
    `https://jsonplaceholder.typicode.com/posts/1`,
    fetcher
  )
  if (error) return 'Failed to load'
  if (!data) return 'Loading...'

  return data.title
}
```

```jsx filename="app/other/page.js" switcher
'use client'

import useSWR from 'swr'

const fetcher = (url) => fetch(url).then((r) => r.json())

export default function Page() {
  const { data, error } = useSWR(
    `https://jsonplaceholder.typicode.com/posts/1`,
    fetcher
  )
  if (error) return 'Failed to load'
  if (!data) return 'Loading...'

  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:

```tsx filename="app/page.tsx" switcher
import Link from 'next/link'

export default function Page() {
  return (
    <>
      <h1>Index Page</h1>
      <hr />
      <ul>
        <li>
          <Link href="/post/1">Post 1</Link>
        </li>
        <li>
~~~

```
        <Link href="/post/2">Post 2</Link>
      </li>
    </ul>
  </>
 )
}
```

```jsx filename="app/page.js" switcher
import Link from 'next/link'

export default function Page() {
  return (
    <>
      <h1>Index Page</h1>
      <p>
        <Link href="/other">Other Page</Link>
      </p>
    </>
  )
}
```

</AppOnly>

<PagesOnly>

## Supported Features

The majority of core Next.js features needed to build a static site are supported, including:

- [Dynamic Routes when using `getStaticPaths`](/docs/app/building-your-application/routing/dynamic-routes)
- Prefetching with `next/link`
- Preloading JavaScript
- [Dynamic Imports](/docs/pages/building-your-application/optimizing/lazy-loading)
- Any styling options (e.g. CSS Modules, styled-jsx)
- [Client-side data fetching](/docs/pages/building-your-application/data-fetching/client-side)
- [`getStaticProps`](/docs/pages/building-your-application/data-fetching/get-static-props)
- [`getStaticPaths`](/docs/pages/building-your-application/data-fetching/get-static-paths)

</PagesOnly>

### Image Optimization

[Image Optimization](/docs/app/building-your-application/optimizing/images) 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 filename="next.config.js"
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export',
  images: {
    loader: 'custom',
    loaderFile: './my-loader.ts',
  },
}

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:

```ts filename="my-loader.ts" switcher
export default function cloudinaryLoader({
  src,
  width,
  quality,
}: {
  src: string
  width: number
  quality?: number
}) {
  const params = ['f_auto', 'c_limit', `w_${width}`, `q_${quality || 'auto'}`]
  return `https://res.cloudinary.com/demo/image/upload/${params.join(
    ','
  )}${src}`
}
```

```js filename="my-loader.js" switcher
export default function cloudinaryLoader({ src, width, quality }) {
  const params = ['f_auto', 'c_limit', `w_${width}`, `q_${quality || 'auto'}`]
  return `https://res.cloudinary.com/demo/image/upload/${params.join(
    ','
  )}${src}`
```

}

You can then use `next/image` in your application, defining relative paths to the image in Cloudinary:

```tsx filename="app/page.tsx" switcher
import Image from 'next/image'

export default function Page() {
  return <Image alt="turtles" src="/turtles.jpg" width={300} height={300} />
}
```

```jsx filename="app/page.js" switcher
import Image from 'next/image'

export default function Page() {
  return <Image alt="turtles" src="/turtles.jpg" width={300} height={300} />
}
```

<AppOnly>

### 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 cached or uncached data. For example:

```ts filename="app/data.json/route.ts" switcher
export async function GET() {
  return Response.json({ name: 'Lee' })
}
```

```js filename="app/data.json/route.js" switcher
export async function GET() {
  return Response.json({ name: 'Lee' })
}
```

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](https://developer.mozilla.org/docs/Web/API) 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:

```jsx
'use client';

import { useEffect } from 'react';

export default function ClientComponent() {
  useEffect(() => {
    // You now have access to `window`
    console.log(window.innerHeight);
  }, [])

  return ...;
}
```

</AppOnly>

## Unsupported Features

Features that require a Node.js server, or dynamic logic that cannot be computed during the build process, are **not** supported:

<AppOnly>

- [Dynamic Routes](/docs/app/building-your-application/routing/dynamic-routes) with `dynamicParams: true`
- [Dynamic Routes](/docs/app/building-your-application/routing/dynamic-routes) without `generateStaticParams()`
- [Route Handlers](/docs/app/building-your-application/routing/route-handlers) that rely on Request
- [Cookies](/docs/app/api-reference/functions/cookies)
- [Rewrites](/docs/app/api-reference/next-config-js/rewrites)
- [Redirects](/docs/app/api-reference/next-config-js/redirects)
- [Headers](/docs/app/api-reference/next-config-js/headers)
- [Middleware](/docs/app/building-your-application/routing/middleware)
- [Incremental Static Regeneration](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating)
- [Image Optimization](/docs/app/building-your-application/optimizing/images) with the default `loader`

- [Draft Mode](/docs/app/building-your-application/configuring/draft-mode)

Attempting to use any of these features with `next dev` will result in an error, similar to setting the [`dynamic`](/docs/app/api-reference/file-conventions/route-segment-config#dynamic) option to `error` in the root layout.

```jsx
export const dynamic = 'error'
```

</AppOnly>

<PagesOnly>

- [Internationalized Routing](/docs/pages/building-your-application/routing/internationalization)
- [API Routes](/docs/pages/building-your-application/routing/api-routes)
- [Rewrites](/docs/pages/api-reference/next-config-js/rewrites)
- [Redirects](/docs/pages/api-reference/next-config-js/redirects)
- [Headers](/docs/pages/api-reference/next-config-js/headers)
- [Middleware](/docs/pages/building-your-application/routing/middleware)
- [Incremental Static Regeneration](/docs/pages/building-your-application/data-fetching/incremental-static-regeneration)
- [Image Optimization](/docs/pages/building-your-application/optimizing/images) with the default `loader`
- [Draft Mode](/docs/pages/building-your-application/configuring/draft-mode)
- [`getStaticPaths` with `fallback: true`](/docs/pages/api-reference/functions/get-static-paths#fallback-true)
- [`getStaticPaths` with `fallback: 'blocking'`](/docs/pages/api-reference/functions/get-static-paths#fallback-blocking)
- [`getServerSideProps`](/docs/pages/building-your-application/data-fetching/get-server-side-props)

</PagesOnly>

## 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. 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 filename="nginx.conf"
server {
  listen 80;
  server_name acme.com;

  root /var/www/out;

  location / {
      try_files $uri $uri.html $uri/ =404;
  }

  # This is necessary when `trailingSlash: false`.
  # You can omit this when `trailingSlash: true`.
  location /blog/ {
      rewrite ^/blog/(.*)$ /blog/$1.html break;
  }

  error_page 404 /404.html;
  location = /404.html {
      internal;
  }
}
```

## Version History

Version	Changes
`v14.0.0`	`next export` has been removed in favor of `"output": "export"`
`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"`

Congratulations, it's time to ship to production.

You can deploy [managed Next.js with Vercel](#managed-nextjs-with-vercel), or self-host on a Node.js server, Docker image, or even static HTML files. When deploying using `next start`, all Next.js features are supported.

## Production Builds

Running `next build` generates an optimized version of your application for production. HTML, CSS, and JavaScript files are created based on your pages. JavaScript is **compiled** and browser bundles are **minified** using the [Next.js Compiler](/docs/architecture/nextjs-compiler) to help achieve the best performance and support [all modern browsers](/docs/architecture/supported-browsers).

Next.js produces a standard deployment output used by managed and self-hosted Next.js. This ensures all features are supported across both methods of deployment. In the next major version, we will be transforming this output into our [Build Output API specification](https://vercel.com/docs/build-output-api/v3?utm_source=next-site&utm_medium=docs&utm_campaign=next-website).

## Managed Next.js with Vercel

[Vercel](https://vercel.com/docs/frameworks/nextjs?utm_source=next-site&utm_medium=docs&utm_campaign=next-website), the creators and maintainers of Next.js, provide managed infrastructure and a developer experience platform for your Next.js applications.

Deploying to Vercel is zero-configuration and provides additional enhancements for scalability, availability, and performance globally. However, all Next.js features are still supported when self-hosted.

Learn more about [Next.js on Vercel](https://vercel.com/docs/frameworks/nextjs?utm_source=next-site&utm_medium=docs&utm_campaign=next-website) or [deploy a template for free](https://vercel.com/templates/next.js?utm_source=next-site&utm_medium=docs&utm_campaign=next-website) to try it out.

## Self-Hosting

You can self-host Next.js in three different ways:

- [A Node.js server](#nodejs-server)
- [A Docker container](#docker-image)
- [A static export](#static-html-export)

### Node.js Server

Next.js can be deployed to any hosting provider that supports Node.js. Ensure your `package.json` has the `"build"` and `"start"` scripts:

```json filename="package.json"
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  }
}
```

Then, run `npm run build` to build your application. Finally, run `npm run start` to start the Node.js server. This server supports all Next.js features.

### Docker Image

Next.js can be deployed to any hosting provider that supports [Docker](https://www.docker.com/) containers. You can use this approach when deploying to container orchestrators such as [Kubernetes](https://kubernetes.io/) or when running inside a container in any cloud provider.

1. [Install Docker](https://docs.docker.com/get-docker/) on your machine
2. [Clone our example](https://github.com/vercel/next.js/tree/canary/examples/with-docker) (or the [multi-environment example](https://github.com/vercel/next.js/tree/canary/examples/with-docker-multi-env))
3. Build your container: `docker build -t nextjs-docker .`
4. Run your container: `docker run -p 3000:3000 nextjs-docker`

Next.js through Docker supports all Next.js features.

### Static HTML Export

Next.js enables starting as a static site or Single-Page Application (SPA), then later optionally upgrading to use features that require a server.

Since Next.js supports this [static export](/docs/app/building-your-application/deploying/static-exports), it can be deployed and hosted on any web server that can serve HTML/CSS/JS static assets. This includes tools like AWS S3, Nginx, or Apache.

Running as a [static export](/docs/app/building-your-application/deploying/static-exports) does not support Next.js features that require a server. [Learn more](/docs/app/building-your-application/deploying/static-exports#unsupported-features).

> **Good to know:**
>
> - [Server Components](/docs/app/building-your-application/rendering/server-components) are supported with static exports.

## Features

### Image Optimization

[Image Optimization](/docs/app/building-your-application/optimizing/images) through `next/image` works self-hosted with zero configuration when deploying using `next start`. If you would prefer to have a separate service to optimize images, you can [configure an image loader](/docs/app/building-your-application/optimizing/images#loaders).

Image Optimization can be used with a [static export](/docs/app/building-your-application/deploying/static-exports#image-optimization) by defining a custom image loader in `next.config.js`. Note that images are optimized at runtime, not during the build.

> **Good to know:**
>
> - When self-hosting, consider installing `sharp` for more performant [Image Optimization](/docs/pages/building-your-application/optimizing/images) in your production environment by running `npm install sharp` in your project directory. On Linux platforms, `sharp` may require [additional configuration](https://sharp.pixelplumbing.com/install#linux-memory-allocator) to prevent excessive memory usage.
> - Learn more about the [caching behavior of optimized images](/docs/app/api-reference/components/image#caching-behavior) and how to configure the TTL.
> - You can also [disable Image Optimization](/docs/app/api-reference/components/image#unoptimized) and still retain other benefits of using `next/image` if you prefer. For example, if you are optimizing images yourself separately.

### Middleware

[Middleware](/docs/app/building-your-application/routing/middleware) works self-hosted with zero configuration when deploying using `next start`. Since it requires access to the incoming request, it is not supported when using a [static export](/docs/app/building-your-application/deploying/static-exports).

Middleware uses a [runtime](/docs/app/building-your-application/rendering/

edge-and-nodejs-runtimes) that is a subset of all available Node.js APIs to help ensure low latency, since it may run in front of every route or asset in your application. This runtime does not require running "at the edge" and works in a single-region server. Additional configuration and infrastructure are required to run Middleware in multiple regions.

If you are looking to add logic (or use an external package) that requires all Node.js APIs, you might be able to move this logic to a [layout](/docs/app/building-your-application/routing/pages-and-layouts#layouts) as a [Server Component](/docs/app/building-your-application/rendering/server-components). For example, checking [headers](/docs/app/api-reference/functions/headers) and [redirecting](/docs/app/api-reference/functions/redirect). You can also use headers, cookies, or query parameters to [redirect](/docs/app/api-reference/next-config-js/redirects#header-cookie-and-query-matching) or [rewrite](/docs/app/api-reference/next-config-js/rewrites#header-cookie-and-query-matching) through `next.config.js`. If that does not work, you can also use a [custom server](/docs/pages/building-your-application/configuring/custom-server).

### Environment Variables

Next.js can support both build time and runtime environment variables.

By default, environment variables are only available on the server. To expose an environment variable to the browser, it must be prefixed with `NEXT_PUBLIC_`. However, these public environment variables will be inlined into the JavaScript bundle during `next build`.

To read runtime environment variables, we recommend using `getServerSideProps` or [incrementally adopting the App Router](/docs/app/building-your-application/upgrading/app-router-migration). With the App Router, we can safely read environment variables on the server during dynamic rendering. This allows you to use a singular Docker image that can be promoted through multiple environments with different values.

```jsx
import { unstable_noStore as noStore } from 'next/cache';

export default function Component() {
  noStore();
  // cookies(), headers(), and other dynamic functions
  // will also opt into dynamic rendering, making
  // this env variable is evaluated at runtime
  const value = process.env.MY_VALUE
  ...
}
```

> **Good to know:**
>
> - You can run code on server startup using the [`register` function](/docs/app/building-your-application/optimizing/instrumentation).
> - We do not recommend using the [runtimeConfig](/docs/pages/api-reference/next-config-js/runtime-configuration) option, as this does not work with the standalone output mode. Instead, we recommend [incrementally adopting](/docs/app/building-your-application/upgrading/app-router-migration) the App Router.

### Caching and ISR

Next.js can cache responses, generated static pages, build outputs, and other static assets like images, fonts, and scripts.

Caching and revalidating pages (using Incremental Static Regeneration (ISR) or newer functions in the App Router) use the **same shared cache**. By default, this cache is stored to the filesystem (on disk) on your Next.js server. **This works automatically when self-hosting** using both the Pages and App Router.

You can configure the Next.js cache location if you want to persist cached pages and data to durable storage, or share the cache across multiple containers or instances of your Next.js application.

#### Automatic Caching

- Next.js sets the `Cache-Control` header of `public, max-age=31536000, immutable` to truly immutable assets. It cannot be overridden. These immutable files contain a SHA-hash in the file name, so they can be safely cached indefinitely. For example, [Static Image Imports](/docs/app/building-your-application/optimizing/images#local-images). You can [configure the TTL](/docs/app/api-reference/components/image#caching-behavior) for images.
- Incremental Static Regeneration (ISR) sets the `Cache-Control` header of `s-maxage: <revalidate in getStaticProps>, stale-while-revalidate`. This revalidation time is defined in your [`getStaticProps` function](/docs/pages/building-your-application/data-fetching/get-static-props) in seconds. If you set `revalidate: false`, it will default to a one-year cache duration.
- Dynamically rendered pages set a `Cache-Control` header of `private, no-cache, no-store, max-age=0, must-revalidate` to prevent user-specific data from being cached. This applies to both the App Router and Pages Router. This also includes [Draft Mode](/docs/app/building-your-application/configuring/draft-mode).

#### Static Assets

If you want to host static assets on a different domain or CDN, you can use the `assetPrefix` [configuration](/docs/app/api-reference/next-config-js/assetPrefix) in `next.config.js`. Next.js will use this asset prefix when retrieving JavaScript or CSS files. Separating your assets to a different domain does come with the downside of extra time spent on DNS and TLS resolution.

[Learn more about `assetPrefix`](/docs/app/api-reference/next-config-js/assetPrefix).

#### Configuring Caching

By default, generated cache assets will be stored in memory (defaults to 50mb) and on disk. If you are hosting Next.js using a container orchestration platform like Kubernetes, each pod will have a copy of the cache. To prevent stale data from being shown since the cache is not shared between pods by default, you can configure the Next.js cache to provide a cache handler and disable in-memory caching.

To configure the ISR/Data Cache location when self-hosting, you can configure a custom handler in your `next.config.js` file:

```jsx filename="next.config.js"
module.exports = {
  experimental: {
    incrementalCacheHandlerPath: require.resolve('./cache-handler.js'),
    isrMemoryCacheSize: 0, // disable default in-memory caching
  },
}
```

Then, create `cache-handler.js` in the root of your project, for example:

```jsx filename="cache-handler.js"
const cache = new Map()

module.exports = class CacheHandler {
  constructor(options) {
    this.options = options
  }

  async get(key) {
    // This could be stored anywhere, like durable storage
    return cache.get(key)
  }

  async set(key, data, ctx) {
    // This could be stored anywhere, like durable storage
    cache.set(key, {
      value: data,
      lastModified: Date.now(),
      tags: ctx.tags,
    })
  }

  async revalidateTag(tag) {
    // Iterate over all entries in the cache
    for (let [key, value] of cache) {
      // If the value's tags include the specified tag, delete this entry
      if (value.tags.includes(tag)) {
        cache.delete(key)
      }
    }
  }
}
```

Using a custom cache handler will allow you to ensure consistency across all pods hosting your Next.js application. For instance, you can save the cached values anywhere, like [Redis](https://github.com/vercel/next.js/tree/canary/examples/cache-handler-redis) or AWS S3.

> **Good to know:**
>
> - `revalidatePath` is a convenience layer on top of cache tags. Calling `revalidatePath` will call the `revalidateTag` function with a special default tag for the provided page.

### Build Cache

Next.js generates an ID during `next build` to identify which version of your application is being served. The same build should be used and boot up multiple containers.

If you are rebuilding for each stage of your environment, you will need to generate a consistent build ID to use between containers. Use the `generateBuildId` command in `next.config.js`:

```jsx filename="next.config.js"
module.exports = {
  generateBuildId: async () => {
    // This could be anything, using the latest git hash
    return process.env.GIT_HASH
  },
```

}

### Version Skew

Next.js will automatically mitigate most instances of [version skew](https://www.industrialempathy.com/posts/version-skew/) and automatically reload the application to retrieve new assets when detected. For example, if there is a mismatch in the build ID, transitions between pages will perform a hard navigation versus using a prefetched value.

When the application is reloaded, there may be a loss of application state if it's not designed to persist between page navigations. For example, using URL state or local storage would persist state after a page refresh. However, component state like `useState` would be lost in such navigations.

Vercel provides additional [skew protection](https://vercel.com/docs/deployments/skew-protection?utm_source=next-site&utm_medium=docs&utm_campaign=next-website) for Next.js applications to ensure assets and functions from the previous build are still available while the new build is being deployed.

<PagesOnly>

## Manual Graceful Shutdowns

When self-hosting, you might want to run code when the server shuts down on `SIGTERM` or `SIGINT` signals.

You can set the env variable `NEXT_MANUAL_SIG_HANDLE` to `true` and then register a handler for that signal inside your `_document.js` file. You will need to register the environment variable directly in the `package.json` script, and not in the `.env` file.

> **Good to know**: Manual signal handling is not available in `next dev`.

```json filename="package.json"
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "NEXT_MANUAL_SIG_HANDLE=true next start"
  }
}
```

```js filename="pages/_document.js"
```

if (process.env.NEXT_MANUAL_SIG_HANDLE) {
  process.on('SIGTERM', () => {
    console.log('Received SIGTERM: cleaning up')
    process.exit(0)
  })
  process.on('SIGINT', () => {
    console.log('Received SIGINT: cleaning up')
    process.exit(0)
  })
}

</PagesOnly>

---
title: Codemods
description: Use codemods to upgrade your Next.js codebase when new features are released.

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:

```bash filename="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

### 14.0

#### Migrate `ImageResponse` imports

##### `next-og-import`

```bash filename="Terminal"
npx @next/codemod@latest next-og-import .
```

This codemod moves transforms imports from `next/server` to `next/og` for usage of [Dynamic OG Image Generation](/docs/app/building-your-application/optimizing/metadata#dynamic-image-generation).

For example:

```js
import { ImageResponse } from 'next/server'
```

Transforms into:

```js
import { ImageResponse } from 'next/og'
```

#### Use `viewport` export

##### `metadata-to-viewport-export`

```bash filename="Terminal"
npx @next/codemod@latest metadata-to-viewport-export .
```

This codemod migrates certain viewport metadata to `viewport` export.

For example:

```js
export const metadata = {
  title: 'My App',
  themeColor: 'dark',
  viewport: {
    width: 1,
  },
}
```

Transforms into:

```js
export const metadata = {
  title: 'My App',
}

export const viewport = {
  width: 1,
  themeColor: 'dark',
}
```

### 13.2

#### Use Built-in Font

##### `built-in-next-font`

```bash filename="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:

```js
import { Inter } from '@next/font/google'
```

Transforms into:

```js
import { Inter } from 'next/font/google'
```

### 13.0

#### Rename Next Image Imports

##### `next-image-to-legacy-image`

```bash filename="Terminal"
npx @next/codemod@latest next-image-to-legacy-image .
```

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:

```jsx filename="pages/index.js"
import Image1 from 'next/image'
import Image2 from 'next/future/image'

export default function Home() {
  return (
    <div>
      <Image1 src="/test.jpg" width="200" height="300" />
      <Image2 src="/test.png" width="500" height="400" />
    </div>
  )
}
```

Transforms into:

```jsx filename="pages/index.js"
// 'next/image' becomes 'next/legacy/image'
import Image1 from 'next/legacy/image'
// 'next/future/image' becomes 'next/image'
import Image2 from 'next/image'

export default function Home() {
  return (
    <div>
      <Image1 src="/test.jpg" width="200" height="300" />
      <Image2 src="/test.png" width="500" height="400" />
    </div>
  )
}
```

#### Migrate to the New Image Component

##### `next-image-experimental`

```bash filename="Terminal"
npx @next/codemod@latest next-image-experimental .
```

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`

```bash filename="Terminal"
npx @next/codemod@latest new-link .
```

<AppOnly>

Remove `<a>` tags inside [Link Components](/docs/app/api-reference/components/link), or add a `legacyBehavior` prop to Links that cannot be auto-fixed.

</AppOnly>

<PagesOnly>

Remove `<a>` tags inside [Link Components](/docs/pages/api-reference/components/link), or add a `legacyBehavior` prop to Links that cannot be auto-fixed.

</PagesOnly>

For example:

```jsx
<Link href="/about">
  <a>About</a>
</Link>
// transforms into
<Link href="/about">
  About
</Link>

<Link href="/about">
  <a onClick={() => console.log('clicked')}>About</a>
```

</Link>
// transforms into
<Link href="/about" onClick={() => console.log('clicked')}>
  About
</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.

```jsx
const Component = () => <a>About</a>

<Link href="/about">
  <Component />
</Link>
// becomes
<Link href="/about" legacyBehavior>
  <Component />
</Link>
```

11

#### Migrate from CRA

##### `cra-to-next`

```bash filename="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](https://github.com/vercel/next.js/discussions/25858).

10

#### Add React imports

##### `add-missing-react-import`

```bash filename="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](https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html) to work.

For example:

```jsx filename="my-component.js"
export default class Home extends React.Component {
  render() {
    return <div>Hello World</div>
  }
}
```

Transforms into:

```jsx filename="my-component.js"
import React from 'react'
export default class Home extends React.Component {
  render() {
    return <div>Hello World</div>
  }
}
```

9

#### Transform Anonymous Components into Named Components

##### `name-default-component`

```bash filename="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](https://nextjs.org/blog/next-9-4#fast-refresh).

For example:

```jsx filename="my-component.js"
export default function () {
  return <div>Hello World</div>
}
```

Transforms into:

```jsx filename="my-component.js"
export default function MyComponent() {
  return <div>Hello World</div>
}
```

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`

```bash filename="Terminal"
npx @next/codemod withamp-to-config
```

Transforms the `withAmp` HOC into Next.js 9 page configuration.

For example:

```js
// Before
import { withAmp } from 'next/amp'

function Home() {
  return <h1>My AMP Page</h1>
}

export default withAmp(Home)
```

```js
// After
export default function Home() {
  return <h1>My AMP Page</h1>
```

}

export const config = {
  amp: *true*,
}

6

#### Use `withRouter`

##### `url-to-withrouter`

```bash filename="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](/docs/messages/url-deprecated)

For example:

```js filename="From"
import React from 'react'
export default class extends React.Component {
  render() {
    const { pathname } = this.props.url
    return <div>Current pathname: {pathname}</div>
  }
}
```

```js filename="To"
import React from 'react'
import { withRouter } from 'next/router'
export default withRouter(
  class extends React.Component {
    render() {
      const { pathname } = this.props.router
      return <div>Current pathname: {pathname}</div>
    }
  }
)
```

This is one case. All the cases that are transformed (and tested) can be found in the [`__testfixtures__` directory](https://github.com/vercel/next.js/tree/canary/packages/next-codemod/transforms/__testfixtures__/url-to-withrouter).

---
title: App Router Incremental Adoption Guide
nav_title: App Router Migration
description: Learn how to upgrade your existing Next.js application from the Pages Router to the App Router.

This guide will help you:

- [Update your Next.js application from version 12 to version 13](#nextjs-version)
- [Upgrade features that work in both the `pages` and the `app` directories](#upgrading-new-features)
- [Incrementally migrate your existing application from `pages` to `app`](#migrating-from-pages-to-app)

## Upgrading

### Node.js Version

The minimum Node.js version is now **v18.17**. See the [Node.js documentation](https://nodejs.org/docs/latest-v18.x/api/) for more information.

### Next.js Version

To update to Next.js version 13, run the following command using your preferred package manager:

```bash filename="Terminal"
npm install next@latest react@latest react-dom@latest
```

### ESLint Version

If you're using ESLint, you need to upgrade your ESLint version:

```bash filename="Terminal"
npm install -D eslint-config-next@latest
```

> **Good to know**: 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](#upgrading-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](#migrating-from-pages-to-app): 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](/docs/app/building-your-application/routing) 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](/docs/app/building-your-application/routing#the-app-router). You can continue using `pages` with new features that work in both directories, such as the updated [Image component](#image-component), [Link component](#link-component), [Script component](#script-component), and [Font optimization](#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**](/docs/app/building-your-application/upgrading/codemods#next-image-to-legacy-image): Safely and automatically renames `next/image` imports to `next/legacy/image`. Existing components will maintain the same behavior.
- [**`next-image-experimental` codemod**](/docs/app/building-your-application/upgrading/codemods#next-image-experimental): 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](/docs/app/building-your-application/routing/linking-and-navigating#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](https://nextjs.org/blog/next-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:

```jsx
import Link from 'next/link'

// Next.js 12: `<a>` has to be nested otherwise it's excluded
<Link href="/about">
  <a>About</a>
</Link>

// Next.js 13: `<Link>` always renders `<a>` under the hood
<Link href="/about">
  About
</Link>
```

To upgrade your links to Next.js 13, you can use the [`new-link` codemod](/docs/app/building-your-application/upgrading/codemods#new-link).

### `<Script>` Component

The behavior of [`next/script`](/docs/app/api-reference/components/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](/docs/app/building-your-application/rendering/server-components) or remove them altogether.

### Font Optimization

Previously, Next.js helped you optimize fonts by [inlining font CSS](/docs/app/building-your-application/optimizing/fonts). Version 13 introduces the new [`next/font`](/docs/app/building-your-application/optimizing/fonts) 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](/docs/app/building-your-application/optimizing/fonts) still works in `pages`, it does not work in `app`. You should use [`next/font`](/docs/app/building-your-application/optimizing/fonts) instead.

See the [Font Optimization](/docs/app/building-your-application/optimizing/fonts) page to learn how to use `next/font`.

## Migrating from `pages` to `app`

> **🎥 Watch:** Learn how to incrementally adopt the App Router → [YouTube (16 minutes)](https://www.youtube.com/watch?v=YQMSietiFm0).

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](/docs/app/building-your-application/routing#file-conventions) and [layouts](/docs/app/building-your-application/routing/pages-and-layouts#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](/docs/app/building-your-application/routing).
- Use nested folders to [define routes](/docs/app/building-your-application/routing/defining-routes) and a special `page.js` file to make a route segment publicly accessible. [Learn more](#step-4-migrating-pages).
- [Special file conventions](/docs/app/building-your-application/routing#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](/docs/app/building-your-application/routing).
- Data fetching functions like `getServerSideProps` and `getStaticProps` have

been replaced with [a new API](/docs/app/building-your-application/data-fetching) inside `app`. `getStaticPaths` has been replaced with [`generateStaticParams`](/docs/app/api-reference/functions/generate-static-params).
- `pages/_app.js` and `pages/_document.js` have been replaced with a single `app/layout.js` root layout. [Learn more](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required).
- `pages/_error.js` has been replaced with more granular `error.js` special files. [Learn more](/docs/app/building-your-application/routing/error-handling).
- `pages/404.js` has been replaced with the [`not-found.js`](/docs/app/api-reference/file-conventions/not-found) file.
- `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):

```bash
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](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required) that will apply to all routes inside `app`.

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  // Layouts must accept a children prop.
  // This will be populated with nested layouts or pages
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
```

```
export default function RootLayout({
  // Layouts must accept a children prop.
  // This will be populated with nested layouts or pages
  children,
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

- 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](/docs/app/building-your-application/optimizing/metadata):

```tsx filename="app/layout.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'Home',
  description: 'Welcome to Next.js',
}
```

```jsx filename="app/layout.js" switcher
export const metadata = {
  title: 'Home',
  description: 'Welcome to Next.js',
}
```

#### 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](/docs/app/building-your-application/rendering/client-components).

#### Migrating the `getLayout()` pattern to Layouts (Optional)

Next.js recommended adding a [property to Page components](/docs/pages/building-your-application/routing/pages-and-layouts#layout-pattern#per-page-layouts) to achieve per-page layouts in the `pages` directory. This pattern can be replaced with native support for [nested layouts](/docs/app/building-your-application/routing/pages-and-layouts#layouts) in the `app` directory.

<details>
  <summary>See before and after example</summary>

Before

```jsx filename="components/DashboardLayout.js"
export default function DashboardLayout({ children }) {
  return (
    <div>
      <h2>My Dashboard</h2>
      {children}
    </div>
  )
}
```

```jsx filename="pages/dashboard/index.js"
import DashboardLayout from '../components/DashboardLayout'

export default function Page() {
  return <p>My Page</p>
}

Page.getLayout = function getLayout(page) {
  return <DashboardLayout>{page}</DashboardLayout>
}
```

After

- Remove the `Page.getLayout` property from `pages/dashboard/index.js` and follow the [steps for migrating pages](#step-4-migrating-pages) to the `app` directory.

```jsx filename="app/dashboard/page.js"
  export default function Page() {
    return <p>My Page</p>
  }
  ```

- Move the contents of `DashboardLayout` into a new [Client Component](/docs/app/building-your-application/rendering/client-components) to retain `pages` directory behavior.

  ```jsx filename="app/dashboard/DashboardLayout.js"
  'use client' // this directive should be at top of the file, before any imports.

  // This is a Client Component
  export default function DashboardLayout({ children }) {
    return (
      <div>
        <h2>My Dashboard</h2>
        {children}
      </div>
    )
  }
  ```

- Import the `DashboardLayout` into a new `layout.js` file inside the `app` directory.

  ```jsx filename="app/dashboard/layout.js"
  import DashboardLayout from './DashboardLayout'

  // This is a Server Component
  export default function Layout({ children }) {
    return <DashboardLayout>{children}</DashboardLayout>
  }
  ```

- You can incrementally move non-interactive parts of `DashboardLayout.js` (Client Component) into `layout.js` (Server Component) to reduce the amount of component JavaScript you send to the client.

</details>

### 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](/docs/app/building-your-application/optimizing/metadata).

Before:

```tsx filename="pages/index.tsx" switcher
import Head from 'next/head'

export default function Page() {
  return (
    <>
      <Head>
        <title>My page title</title>
      </Head>
    </>
  )
}
```

```jsx filename="pages/index.js" switcher
import Head from 'next/head'

export default function Page() {
  return (
    <>
      <Head>
        <title>My page title</title>
      </Head>
    </>
  )
}
```

After:

```tsx filename="app/page.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'My Page Title',
}

export default function Page() {
  return '...'
}
```

```jsx filename="app/page.js" switcher
export const metadata = {
  title: 'My Page Title',
}

export default function Page() {
  return '...'
}
```

[See all metadata options](/docs/app/api-reference/functions/generate-metadata).

### Step 4: Migrating Pages

- Pages in the [`app` directory](/docs/app/building-your-application/routing) are [Server Components](/docs/app/building-your-application/rendering/server-components) by default. This is different from the `pages` directory where pages are [Client Components](/docs/app/building-your-application/rendering/client-components).
- [Data fetching](/docs/app/building-your-application/data-fetching) has changed in `app`. `getServerSideProps`, `getStaticProps` and `getInitialProps` have been replaced with a simpler API.
- The `app` directory uses nested folders to [define routes](/docs/app/building-your-application/routing/defining-routes) and a special `page.js` file to make a route segment publicly accessible.
`pages` Directory	`app` Directory	Route
`index.js`	`page.js`	`/`
`about.js`	`about/page.js`	`/about`
`blog/[slug].js`	`blog/[slug]/page.js`	`/blog/post-1`

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.

> **Good to know**: 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`.

```tsx filename="app/home-page.tsx" switcher
'use client'

// This is a Client Component. It receives data as props and
// has access to state and effects just like Page components
// in the `pages` directory.
export default function HomePage({ recentPosts }) {
  return (
    <div>
      {recentPosts.map((post) => (
        <div key={post.id}>{post.title}</div>
      ))}
    </div>
  )
}
```

```jsx filename="app/home-page.js" switcher
'use client'

// This is a Client Component. It receives data as props and
// has access to state and effects just like Page components
// in the `pages` directory.
export default function HomePage({ recentPosts }) {
  return (
    <div>
      {recentPosts.map((post) => (
        <div key={post.id}>{post.title}</div>
      ))}
    </div>
  )
}
```

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](/docs/app/building-your-application/data-fetching/fetching-caching-and-

revalidating). See the [data fetching upgrade guide](#step-6-migrating-data-fetching-methods) for more details.

  ```tsx filename="app/page.tsx" switcher
  // Import your Client Component
  import HomePage from './home-page'

  async function getPosts() {
    const res = await fetch('https://...')
    const posts = await res.json()
    return posts
  }

  export default async function Page() {
    // Fetch data directly in a Server Component
    const recentPosts = await getPosts()
    // Forward fetched data to your Client Component
    return <HomePage recentPosts={recentPosts} />
  }
  ```

  ```jsx filename="app/page.js" switcher
  // Import your Client Component
  import HomePage from './home-page'

  async function getPosts() {
    const res = await fetch('https://...')
    const posts = await res.json()
    return posts
  }

  export default async function Page() {
    // Fetch data directly in a Server Component
    const recentPosts = await getPosts()
    // Forward fetched data to your Client Component
    return <HomePage recentPosts={recentPosts} />
  }
  ```

- If your previous page used `useRouter`, you'll need to update to the new routing hooks. [Learn more](/docs/app/api-reference/functions/use-router).
- Start your development server and visit [`http://localhost:3000`](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()`](/docs/app/api-reference/functions/use-router), [`usePathname()`](/docs/app/api-reference/functions/use-pathname), and [`useSearchParams()`](/docs/app/api-reference/functions/use-search-params).

- 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`](/docs/pages/api-reference/functions/use-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](/docs/app/api-reference/functions/use-router#router-events) section for more details.
- These new hooks are only supported in Client Components. They cannot be used in Server Components.

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { useRouter, usePathname, useSearchParams } from 'next/navigation'

export default function ExampleClientComponent() {
  const router = useRouter()
  const pathname = usePathname()
  const searchParams = useSearchParams()

  // ...
}
```

```jsx filename="app/example-client-component.js" switcher
'use client'

import { useRouter, usePathname, useSearchParams } from 'next/navigation'

export default function ExampleClientComponent() {
  const router = useRouter()
```

  const pathname = usePathname()
  const searchParams = useSearchParams()

  // ...
}
~~~

In addition, the new `useRouter` hook has the following changes:

- `isFallback` has been removed because `fallback` has [been replaced](#replacing-fallback).
- 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](/docs/pages/building-your-application/routing/internationalization).
- `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](/docs/app/building-your-application/rendering/server-components#static-rendering-default), any component that uses the [`useSearchParams()`](/docs/app/api-reference/functions/use-search-params) hook will skip the prerendering step and instead be rendered on the client at runtime.

[View the `useRouter()` API reference](/docs/app/api-reference/functions/use-router).

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](/docs/app/building-your-application/data-fetching) built on top of `fetch()` and `async` React Server Components.

```tsx filename="app/page.tsx" switcher
export default async function Page() {
  // This request should be cached until manually invalidated.
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be omitted.
  const staticData = await fetch(`https://...`, { cache: 'force-cache' })

  // This request should be refetched on every request.
  // Similar to `getServerSideProps`.
  const dynamicData = await fetch(`https://...`, { cache: 'no-store' })
~~~

```
  // This request should be cached with a lifetime of 10 seconds.
  // Similar to `getStaticProps` with the `revalidate` option.
  const revalidatedData = await fetch(`https://...`, {
    next: { revalidate: 10 },
  })

  return <div>...</div>
}
```

```jsx filename="app/page.js" switcher
export default async function Page() {
  // This request should be cached until manually invalidated.
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be omitted.
  const staticData = await fetch(`https://...`, { cache: 'force-cache' })

  // This request should be refetched on every request.
  // Similar to `getServerSideProps`.
  const dynamicData = await fetch(`https://...`, { cache: 'no-store' })

  // This request should be cached with a lifetime of 10 seconds.
  // Similar to `getStaticProps` with the `revalidate` option.
  const revalidatedData = await fetch(`https://...`, {
    next: { revalidate: 10 },
  })

  return <div>...</div>
}
```

#### 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).

```jsx filename="pages/dashboard.js"
// `pages` directory

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

  return { props: { projects } }
```

}

export default function Dashboard({ projects }) {
  return (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
)
}

In the `app` directory, we can colocate our data fetching inside our React components using [Server Components](/docs/app/building-your-application/rendering/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](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating). This is similar to `getServerSideProps` in the `pages` directory.

```tsx filename="app/dashboard/page.tsx" switcher
// `app` directory

// This function can be named anything
async function getProjects() {
  const res = await fetch(`https://...`, { cache: 'no-store' })
  const projects = await res.json()

  return projects
}

export default async function Dashboard() {
  const projects = await getProjects()

  return (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
  )
}
```

```jsx filename="app/dashboard/page.js" switcher
// `app` directory

// This function can be named anything
async function getProjects() {
  const res = await fetch(`https://...`, { cache: 'no-store' })
  const projects = await res.json()

  return projects
}

export default async function Dashboard() {
  const projects = await getProjects()

  return (
    <ul>
      {projects.map((project) => (
        <li key={project.id}>{project.name}</li>
      ))}
    </ul>
  )
}
```

#### 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.

```jsx filename="pages/index.js"
// `pages` directory

export async function getServerSideProps({ req, query }) {
  const authHeader = req.getHeaders()['authorization'];
  const theme = req.cookies['theme'];

  return { props: { ... }}
}

export default function Page(props) {
  return ...
}
```

The `app` directory exposes new read-only functions to retrieve request data:

- [`headers()`](/docs/app/api-reference/functions/headers): Based on the Web Headers API, and can be used inside [Server Components](/docs/app/building-your-application/rendering/server-components) to retrieve request headers.
- [`cookies()`](/docs/app/api-reference/functions/cookies): Based on the Web Cookies API, and can be used inside [Server Components](/docs/app/building-your-application/rendering/server-components) to retrieve cookies.

```tsx filename="app/page.tsx" switcher
// `app` directory
import { cookies, headers } from 'next/headers'

async function getData() {
  const authHeader = headers().get('authorization')

  return '...'
}

export default async function Page() {
  // You can use `cookies()` or `headers()` inside Server Components
  // directly or in your data fetching function
  const theme = cookies().get('theme')
  const data = await getData()
  return '...'
}
```

```jsx filename="app/page.js" switcher
// `app` directory
import { cookies, headers } from 'next/headers'

async function getData() {
  const authHeader = headers().get('authorization')

  return '...'
}

export default async function Page() {
  // You can use `cookies()` or `headers()` inside Server Components
  // directly or in your data fetching function
  const theme = cookies().get('theme')
  const data = await getData()
  return '...'
}
```

#### 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.

```jsx filename="pages/index.js"
// `pages` directory

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

  return { props: { projects } }
}

export default function Index({ projects }) {
  return projects.map((project) => <div>{project.name}</div>)
}
```

In the `app` directory, data fetching with [`fetch()`](/docs/app/api-reference/functions/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.

```jsx filename="app/page.js"
// `app` directory

// This function can be named anything
async function getProjects() {
  const res = await fetch(`https://...`)
  const projects = await res.json()

  return projects
}

export default async function Index() {
  const projects = await getProjects()

  return projects.map((project) => <div>{project.name}</div>)
}
```

#### 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.

```jsx filename="pages/posts/[id].js"
// `pages` directory
import PostLayout from '@/components/post-layout'

export async function getStaticPaths() {
  return {
    paths: [{ params: { id: '1' } }, { params: { id: '2' } }],
  }
}

export async function getStaticProps({ params }) {
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  return { props: { post } }
}

export default function Post({ post }) {
  return <PostLayout post={post} />
}
```

In the `app` directory, `getStaticPaths` is replaced with [`generateStaticParams`](/docs/app/api-reference/functions/generate-static-params).

[`generateStaticParams`](/docs/app/api-reference/functions/generate-static-params) behaves similarly to `getStaticPaths`, but has a simplified API for returning route parameters and can be used inside [layouts](/docs/app/building-your-application/routing/pages-and-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.

```jsx filename="app/posts/[id]/page.js"
// `app` directory
import PostLayout from '@/components/post-layout'

export async function generateStaticParams() {
  return [{ id: '1' }, { id: '2' }]
}

async function getPost(params) {
  const res = await fetch(`https://.../posts/${params.id}`)
  const post = await res.json()

  return post
}

export default async function Post({ params }) {
  const post = await getPost(params)

  return <PostLayout post={post} />
}
```

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.

```jsx filename="pages/posts/[id].js"
// `pages` directory

export async function getStaticPaths() {
  return {
    paths: [],
    fallback: 'blocking'
  };
}

export async function getStaticProps({ params }) {
  ...
}

export default function Post({ post }) {
  return ...
}
```

In the `app` directory the [`config.dynamicParams` property](/docs/app/api-reference/file-conventions/route-segment-config#dynamicparams) controls how params outside of [`generateStaticParams`](/docs/app/api-reference/functions/generate-static-params) 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.

```jsx filename="app/posts/[id]/page.js"
// `app` directory

export const dynamicParams = true;

export async function generateStaticParams() {
  return [...]
}

async function getPost(params) {
  ...
}

export default async function Post({ params }) {
  const post = await getPost(params);

  return ...
}
```

With [`dynamicParams`](/docs/app/api-reference/file-conventions/route-segment-config#dynamicparams) set to `true` (the default), when a route segment is requested that hasn't been generated, it will be server-rendered and cached.

#### 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.

```jsx filename="pages/index.js"
// `pages` directory

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

  return {
    props: { posts },
    revalidate: 60,
  }
}

export default function Index({ posts }) {
  return (
    <Layout>
      <PostList posts={posts} />
    </Layout>
  )
}
```

In the `app` directory, data fetching with [`fetch()`](/docs/app/api-reference/functions/fetch) can use `revalidate`, which will cache the request for the specified amount of seconds.

```jsx filename="app/page.js"
// `app` directory

async function getPosts() {
  const res = await fetch(`https://.../posts`, { next: { revalidate: 60 } })
  const data = await res.json()

  return data.posts
}

export default async function PostList() {
  const posts = await getPosts()

  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](/docs/app/building-your-application/routing/route-handlers) in the `app` directory.

Route Handlers allow you to create custom request handlers for a given route using the Web [Request](https://developer.mozilla.org/docs/Web/API/Request) and [Response](https://developer.mozilla.org/docs/Web/API/Response) APIs.

```ts filename="app/api/route.ts" switcher
export async function GET(request: Request) {}
```

```js filename="app/api/route.js" switcher
export async function GET(request) {}
```

> **Good to know**: If you previously used API routes to call an external API from the client, you can now use [Server Components](/docs/app/building-your-application/rendering/server-components) instead to securely fetch data. Learn more about [data fetching](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating).

### 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](/docs/app/building-your-application/styling/css-modules)
- [Tailwind CSS](/docs/app/building-your-application/styling/tailwind-css)
- [Global Styles](/docs/app/building-your-application/styling/css-modules#global-styles)
- [CSS-in-JS](/docs/app/building-your-application/styling/css-in-js)
- [External Stylesheets](/docs/app/building-your-application/styling/css-modules#external-stylesheets)
- [Sass](/docs/app/building-your-application/styling/sass)

#### Tailwind CSS

If you're using Tailwind CSS, you'll need to add the `app` directory to your `tailwind.config.js` file:

```js filename="tailwind.config.js"
module.exports = {
  content: [
    './app/**/*.{js,ts,jsx,tsx,mdx}', // <-- Add this line
    './pages/**/*.{js,ts,jsx,tsx,mdx}',
    './components/**/*.{js,ts,jsx,tsx,mdx}',
```

  ],
}

You'll also need to import your global styles in your `app/layout.js` file:

```jsx filename="app/layout.js"
import '../styles/globals.css'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

Learn more about [styling with Tailwind CSS](/docs/app/building-your-application/styling/tailwind-css)

## Codemods

Next.js provides Codemod transformations to help upgrade your codebase when a feature is deprecated. See [Codemods](/docs/app/building-your-application/upgrading/codemods) for more information.

---
title: Version 14
description: Upgrade your Next.js Application from Version 13 to 14.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

## Upgrading from 13 to 14

To update to Next.js version 14, run the following command using your preferred package manager:

```bash filename="Terminal"
npm i next@latest react@latest react-dom@latest eslint-config-next@latest
```

```bash filename="Terminal"
```

yarn add next@latest react@latest react-dom@latest eslint-config-next@latest

```bash filename="Terminal"
pnpm up next react react-dom eslint-config-next --latest
```

```bash filename="Terminal"
bun add next@latest react@latest react-dom@latest eslint-config-next@latest
```

> **Good to know:** If you are using TypeScript, ensure you also upgrade `@types/react` and `@types/react-dom` to their latest versions.

### v14 Summary

- The minimum Node.js version has been bumped from 16.14 to 18.17, since 16.x has reached end-of-life.
- The `next export` command is deprecated in favor of `output: 'export'`. Please see the [docs](https://nextjs.org/docs/app/building-your-application/deploying/static-exports) for more information.
- The `next/server` import for `ImageResponse` was renamed to `next/og`. A [codemod is available](/docs/app/building-your-application/upgrading/codemods#next-og-import) to safely and automatically rename your imports.
- The `@next/font` package has been fully removed in favor of the built-in `next/font`. A [codemod is available](/docs/app/building-your-application/upgrading/codemods#built-in-next-font) to safely and automatically rename your imports.
- The WASM target for `next-swc` has been removed.

---
title: Migrating from Vite
description: Learn how to migrate your existing React application from Vite to Next.js.

This guide will help you migrate an existing Vite application to Next.js.

## Why Switch?

There are several reasons why you might want to switch from Vite to Next.js:

1. **Slow initial page loading time**: If you have built your application with the [default Vite
   plugin for React](https://github.com/vitejs/vite-plugin-react/tree/main/packages/plugin-react),
   your application is a purely client-side application. Client-side only

applications, also known as
   single-page applications (SPAs), often experience slow initial page loading time. This
   happens due to a couple of reasons:
   1. The browser needs to wait for the React code and your entire application bundle to download
      and run before your code is able to send requests to load some data.
   2. Your application code grows with every new feature and extra dependency you add.
2. **No automatic code splitting**: The previous issue of slow loading times can be somewhat managed with code splitting.
   However, if you try to do code splitting manually, you'll often make performance worse. It's easy
   to inadvertently introduce network waterfalls when code-splitting manually. Next.js provides
   automatic code splitting built into its router.
3. **Network waterfalls**: A common cause of poor performance occurs when applications make
   sequential client-server requests to fetch data. One common pattern for data fetching in an SPA
   is to initially render a placeholder, and then fetch data after the component has mounted.
   Unfortunately, this means that a child component that fetches data can't start fetching until
   the parent component has finished loading its own data. On Next.js,
   [this issue is resolved](https://github.com/reactjs/rfcs/blob/main/text/0188-server-components.md#no-client-server-waterfalls)
   by fetching data in Server Components.
4. **Fast and intentional loading states**: Thanks to built-in support for
   [Streaming with Suspense](/docs/app/building-your-application/routing/loading-ui-and-streaming#streaming-with-suspense),
   with Next.js, you can be more intentional about which parts of your UI you want to load first and
   in what order without introducing network waterfalls. This enables you to build pages that are
   faster to load and also eliminate [layout shifts](https://web.dev/cls/).
5. **Choose the data fetching strategy**: Depending on your needs, Next.js allows you to choose your
   data fetching strategy on a page and component basis. You can decide to fetch at build time, at
   request time on the server, or on the client. For example, you can fetch data from your CMS and
   render your blog posts at build time, which can then be efficiently cached on a CDN.
6. **Middleware**: [Next.js Middleware](/docs/app/building-your-application/routing/middleware)
   allows you to run code on the server before a request is completed. This is

especially useful to
   avoid having a flash of unauthenticated content when the user visits an authenticated-only page
   by redirecting the user to a login page. The middleware is also useful for experimentation and
   internationalization.
7. **Built-in Optimizations**: Images, fonts, and third-party scripts often have significant impact
   on an application's performance. Next.js comes with built-in components that automatically
   optimize those for you.

## Migration Steps

Our goal with this migration is to get a working Next.js application as quickly as possible, so that
you can then adopt Next.js features incrementally. To begin with, we'll keep it as a purely
client-side application (SPA) without migrating your existing router. This helps minimize the
chances of encountering issues during the migration process and reduces merge conflicts.

### Step 1: Install the Next.js Dependency

The first thing you need to do is to install `next` as a dependency:

```bash filename="Terminal"
npm install next@latest
```

### Step 2: Create the Next.js Configuration File

Create a `next.config.mjs` at the root of your project. This file will hold your
[Next.js configuration options](/docs/app/api-reference/next-config-js).

```js filename="next.config.mjs"
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export', // Outputs a Single-Page Application (SPA).
  distDir: './dist', // Changes the build output directory to `./dist/`.
}

export default nextConfig
```

> **Good to know:** You can use either `.js` or `.mjs` for your Next.js

configuration file.

### Step 3: Update TypeScript Configuration

If you're using TypeScript, you need to update your `tsconfig.json` file with the following changes
to make it compatible with Next.js. If you're not using TypeScript, you can skip this step.

1. Remove the [project reference](https://www.typescriptlang.org/tsconfig#references) to `tsconfig.node.json`
2. Add `./dist/types/**/*.ts` and `./next-env.d.ts` to the [`include` array](https://www.typescriptlang.org/tsconfig#include)
3. Add `./node_modules` to the [`exclude` array](https://www.typescriptlang.org/tsconfig#exclude)
4. Add `{ "name": "next" }` to the [`plugins` array in `compilerOptions`](https://www.typescriptlang.org/tsconfig#plugins): `"plugins": [{ "name": "next" }]`
5. Set [`esModuleInterop`](https://www.typescriptlang.org/tsconfig#esModuleInterop) to `true`: `"esModuleInterop": true`
6. Set [`jsx`](https://www.typescriptlang.org/tsconfig#jsx) to `preserve`: `"jsx": "preserve"`
7. Set [`allowJs`](https://www.typescriptlang.org/tsconfig#allowJs) to `true`: `"allowJs": true`
8. Set [`forceConsistentCasingInFileNames`](https://www.typescriptlang.org/tsconfig#forceConsistentCasingInFileNames) to `true`: `"forceConsistentCasingInFileNames": true`
9. Set [`incremental`](https://www.typescriptlang.org/tsconfig#incremental) to `true`: `"incremental": true`

Here's an example of a working `tsconfig.json` with those changes:

```json filename="tsconfig.json"
{
  "compilerOptions": {
    "target": "ES2020",
    "useDefineForClassFields": true,
    "lib": ["ES2020", "DOM", "DOM.Iterable"],
    "module": "ESNext",
    "esModuleInterop": true,
    "skipLibCheck": true,
    "moduleResolution": "bundler",
    "allowImportingTsExtensions": true,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "preserve",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noFallthroughCasesInSwitch": true,
    "allowJs": true,
    "forceConsistentCasingInFileNames": true,
    "incremental": true,
    "plugins": [{ "name": "next" }]
  },
  "include": ["./src", "./dist/types/**/*.ts", "./next-env.d.ts"],
  "exclude": ["./node_modules"]
}
```

You can find more information about configuring TypeScript on the
[Next.js docs](/docs/app/building-your-application/configuring/typescript#typescript-plugin).

### Step 4: Create the Root Layout

A Next.js [App Router](/docs/app) application must include a
[root layout](/docs/app/building-your-application/routing/pages-and-layouts#root-layout-required)
file, which is a [React Server Component](/docs/app/building-your-application/rendering/server-components)
that will wrap all pages in your application. This file is defined at the top level of the `app`
directory.

The closest equivalent to the root layout file in a Vite application is the
[`index.html` file](https://vitejs.dev/guide/#index-html-and-project-root), which contains your
`<html>`, `<head>`, and `<body>` tags.

In this step, you'll convert your `index.html` file into a root layout file:

1. Create a new `app` directory in your `src` directory.
2. Create a new `layout.tsx` file inside that `app` directory:

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return null
}
```

```jsx filename="app/layout.js" switcher
export default function RootLayout({ children }) {
  return null
}
```

> **Good to know**: `.js`, `.jsx`, or `.tsx` extensions can be used for Layout files.

3. Copy the content of your `index.html` file into the previously created `<RootLayout>` component while
   replacing the `body.div#root` and `body.script` tags with `<div id="root">{children}</div>`:

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <head>
        <meta charset="UTF-8" />
        <link rel="icon" type="image/svg+xml" href="/icon.svg" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" /
>
        <title>My App</title>
        <meta name="description" content="My App is a..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <head>
        <meta charset="UTF-8" />
        <link rel="icon" type="image/svg+xml" href="/icon.svg" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" /
>
        <title>My App</title>
        <meta name="description" content="My App is a..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

4. Next.js already includes by default the
   [meta charset](https://developer.mozilla.org/docs/Web/HTML/Element/meta#charset) and
   [meta viewport](https://developer.mozilla.org/docs/Web/HTML/Viewport_meta_tag) tags, so you
   can safely remove those from your `<head>`:

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <head>
        <link rel="icon" type="image/svg+xml" href="/icon.svg" />
        <title>My App</title>
        <meta name="description" content="My App is a..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <head>
        <link rel="icon" type="image/svg+xml" href="/icon.svg" />
        <title>My App</title>
        <meta name="description" content="My App is a..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

5. Any [metadata files](/docs/app/building-your-application/optimizing/metadata#file-based-metadata)
   such as `favicon.ico`, `icon.png`, `robots.txt` are automatically added to the application
   `<head>` tag as long as you have them placed into the top level of the `app` directory. After
   moving
   [all supported files](/docs/app/building-your-application/optimizing/metadata#file-based-metadata)
   into the `app` directory you can safely delete their `<link>` tags:

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <head>
        <title>My App</title>
        <meta name="description" content="My App is a..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <head>
```

```
        <title>My App</title>
        <meta name="description" content="My App is a..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

6. Finally, Next.js can manage your last `<head>` tags with the
   [Metadata API](/docs/app/building-your-application/optimizing/metadata).
Move your final metadata
   info into an exported
   [`metadata` object](/docs/app/api-reference/functions/generate-metadata#metadata-object):

```tsx filename="app/layout.tsx" switcher
import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'My App',
  description: 'My App is a...',
}

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
export const metadata = {
  title: 'My App',
  description: 'My App is a...',
}
```

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

With the above changes, you shifted from declaring everything in your
`index.html` to using Next.js'
convention-based approach built into the framework
([Metadata API](/docs/app/building-your-application/optimizing/metadata)).
This approach enables you
to more easily improve your SEO and web shareability of your pages.

### Step 5: Create the Entrypoint Page

On Next.js you declare an entrypoint for your application by creating a
`page.tsx` file. The
closest equivalent of this file on Vite is your `main.tsx` file. In this step, you’ll
set up the
entrypoint of your application.

1. **Create a `[[...slug]]` directory in your `app` directory.**

Since in this guide we're aiming first to set up our Next.js as an SPA (Single
Page Application),
you need your page entrypoint to catch all possible routes of your application.
For that, create a
new `[[...slug]]` directory in your `app` directory.

This directory is what is called an
[optional catch-all route segment](/docs/app/building-your-application/routing/dynamic-routes#optional-catch-all-segments).
Next.js uses a file-system based router where
[directories are used to define routes](/docs/app/building-your-application/routing/defining-routes#creating-routes).
This special directory will make sure that all routes of your application will be
directed to its
containing `page.tsx` file.

2. **Create a new `page.tsx` file inside the `app/[[...slug]]` directory with
the following content:**

```tsx filename="app/[[...slug]]/page.tsx" switcher
'use client'

import dynamic from 'next/dynamic'
import '../../index.css'

const App = dynamic(() => import('../../App'), { ssr: false })

export default function Page() {
  return <App />
}
```

```jsx filename="app/[[...slug]]/page.js" switcher
'use client'

import dynamic from 'next/dynamic'
import '../../index.css'

const App = dynamic(() => import('../../App'), { ssr: false })

export default function Page() {
  return <App />
}
```

> **Good to know**: `.js`, `.jsx`, or `.tsx` extensions can be used for Page
files.

This file contains a `<Page>` component which is marked as a
[Client Component](/docs/app/building-your-application/rendering/client-components) by the `'use client'`
directive. Without that directive, the component would have been a
[Server Component](/docs/app/building-your-application/rendering/server-components).

In Next.js, Client Components are [prerendered to HTML](/docs/app/building-your-application/rendering/client-components#how-are-client-components-rendered)
on the server before being sent to the client, but since we want to first have a
purely client-side
application, you need to tell Next.js to disable the prerendering for the `<App>`
component by
dynamically importing it with the `ssr` option set to `false`:

```tsx
```

const App = dynamic(() => import('../../App'), { ssr: *false* })

### Step 6: Update Static Image Imports

Next.js handles static image imports slightly different from Vite. With Vite, importing an image
file will return its public URL as a string:

```tsx filename="App.tsx"
import image from './img.png' // `image` will be '/assets/img.2d8efhg.png' in production

export default function App() {
  return <img src={image} />
}
```

With Next.js, static image imports return an object. The object can then be used directly with the
Next.js [`<Image>` component](/docs/app/api-reference/components/image), or you can use the object's
`src` property with your existing `<img>` tag.

The `<Image>` component has the added benefits of
[automatic image optimization](/docs/app/building-your-application/optimizing/images). The `<Image>`
component automatically sets the `width` and `height` attributes of the resulting `<img>` based on
the image's dimensions. This prevents layout shifts when the image loads. However, this can cause
issues if your app contains images with only one of their dimensions being styled without the other
styled to `auto`. When not styled to `auto`, the dimension will default to the `<img>` dimension
attribute's value, which can cause the image to appear distorted.

Keeping the `<img>` tag will reduce the amount of changes in your application and prevent the above
issues. However, you'll still want to later migrate to the `<Image>` component to take advantage of
the automatic optimizations.

1. **Convert absolute import paths for images imported from `/public` into relative imports:**

```tsx
```

// Before
import logo from '/logo.png'

// After
import logo from '../public/logo.png'

2. **Pass the image `src` property instead of the whole image object to your `<img>` tag:**

```tsx
// Before
<img src={logo} />

// After
<img src={logo.src} />
```

> **Warning:** If you're using TypeScript, you might encounter type errors when accessing the `src`
> property. You can safely ignore those for now. They will be fixed by the end of this guide.

### Step 7: Migrate the Environment Variables

Next.js has support for `.env`
[environment variables](/docs/app/building-your-application/configuring/environment-variables)
similar to Vite. The main difference is the prefix used to expose environment variables on the
client-side.

- Change all environment variables with the `VITE_` prefix to `NEXT_PUBLIC_`.

Vite exposes a few built-in environment variables on the special `import.meta.env` object which
aren’t supported by Next.js. You need to update their usage as follows:

- `import.meta.env.MODE` ⇒ `process.env.NODE_ENV`
- `import.meta.env.PROD` ⇒ `process.env.NODE_ENV === 'production'`
- `import.meta.env.DEV` ⇒ `process.env.NODE_ENV !== 'production'`
- `import.meta.env.SSR` ⇒ `typeof window !== 'undefined'`

Next.js also doesn't provide a built-in `BASE_URL` environment variable. However, you can still
configure one, if you need it:

1. **Add the following to your `.env` file:**

```bash filename=".env"
# ...
NEXT_PUBLIC_BASE_PATH="/some-base-path"
```

2. **Set [`basePath`](/docs/app/api-reference/next-config-js/basePath) to `process.env.NEXT_PUBLIC_BASE_PATH` in your `next.config.mjs` file:**

```js filename="next.config.mjs"
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export', // Outputs a Single-Page Application (SPA).
  distDir: './dist', // Changes the build output directory to `./dist/`.
  basePath: process.env.NEXT_PUBLIC_BASE_PATH, // Sets the base path to `/some-base-path`.
}

export default nextConfig
```

3. **Update `import.meta.env.BASE_URL` usages to `process.env.NEXT_PUBLIC_BASE_PATH`**

### Step 8: Update Scripts in `package.json`

You should now be able to run your application to test if you successfully migrated to Next.js. But
before that, you need to update your `scripts` in your `package.json` with Next.js related commands,
and add `.next` and `next-env.d.ts` to your `.gitignore`:

```json filename="package.json"
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  }
}
```

```txt filename=".gitignore"
# ...
.next
```

next-env.d.ts
```

Now run `npm run dev`, and open [`http://localhost:3000`](http://localhost:3000). You should
hopefully see your application now running on Next.js.

If your application followed a conventional Vite configuration, this is all you would need to do
to have a working version of your application.

> **Example:** Check out [this pull request](https://github.com/inngest/vite-to-nextjs/pull/1) for a
> working example of a Vite application migrated to Next.js.

Step 9: Clean Up

You can now clean up your codebase from Vite related artifacts:

- Delete `main.tsx`
- Delete `index.html`
- Delete `vite-env.d.ts`
- Delete `tsconfig.node.json`
- Delete `vite.config.ts`
- Uninstall Vite dependencies

Next Steps

If everything went according to plan, you now have a functioning Next.js application running as a
single-page application. However, you aren't yet taking advantage of most of Next.js' benefits, but
you can now start making incremental changes to reap all the benefits. Here's what you might want to
do next:

- Migrate from React Router to the [Next.js App Router](/docs/app/building-your-application/routing) to get:
  - Automatic code splitting
  - [Streaming Server-Rendering](/docs/app/building-your-application/routing/loading-ui-and-streaming)
  - [React Server Components](/docs/app/building-your-application/rendering/server-components)
- [Optimize images with the `<Image>` component](/docs/app/building-your-application/optimizing/images)
- [Optimize fonts with `next/font`](/docs/app/building-your-application/optimizing/fonts)
```

- [Optimize third-party scripts with the `<Script>` component](/docs/app/building-your-application/optimizing/scripts)
- [Update your ESLint configuration to support Next.js rules](/docs/app/building-your-application/configuring/eslint)

---
title: Upgrade Guide
nav_title: Upgrading
description: Learn how to upgrade to the latest versions of Next.js.
---

Upgrade your application to newer versions of Next.js or migrate from the Pages Router to the App Router.

---
title: Building Your Application
description: Learn how to use Next.js features to build your application.
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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.

<AppOnly>

If you're new to Next.js, we recommend starting with the [Routing](/docs/app/building-your-application/routing), [Rendering](/docs/app/building-your-application/rendering), [Data Fetching](/docs/app/building-your-application/data-fetching) and [Styling](/docs/app/building-your-application/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](/docs/app/building-your-application/optimizing) and [Configuring](/docs/app/building-your-application/configuring). Finally, once you're ready, checkout the [Deploying](/docs/app/building-your-application/deploying) and [Upgrading](/docs/app/building-your-application/upgrading) sections.

</AppOnly>

<PagesOnly>

If you're new to Next.js, we recommend starting with the [Routing](/docs/pages/building-your-application/routing), [Rendering](/docs/pages/building-your-application/rendering), [Data Fetching](/docs/pages/building-your-application/data-fetching) and [Styling](/docs/pages/building-your-application/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](/docs/pages/building-your-application/optimizing) and [Configuring](/docs/pages/building-your-application/configuring). Finally, once you're ready, checkout the [Deploying](/docs/pages/building-your-application/deploying) and [Upgrading](/docs/pages/building-your-application/upgrading) sections.

</PagesOnly>

---
title: <Script>
description: Optimize third-party scripts in your Next.js application using the built-in `next/script` Component.
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

This API reference will help you understand how to use [props](#props) available for the Script Component. For features and usage, please see the [Optimizing Scripts](/docs/app/building-your-application/optimizing/scripts) page.

```tsx filename="app/dashboard/page.tsx" switcher
import Script from 'next/script'

export default function Dashboard() {
  return (
    <>
      <Script src="https://example.com/script.js" />
    </>
  )
}
```

```jsx filename="app/dashboard/page.js" switcher
import Script from 'next/script'
```

```
export default function Dashboard() {
  return (
    <>
      <Script src="https://example.com/script.js" />
    </>
  )
}
```

## Props

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

Prop	Example	Type	Required
[`src`](#src)	`src="http://example.com/script"`	String	Required unless inline script is used
[`strategy`](#strategy)	`strategy="lazyOnload"`	String	-
[`onLoad`](#onload)	`onLoad={onLoadFunc}`	Function	-
[`onReady`](#onready)	`onReady={onReadyFunc}`	Function	-
[`onError`](#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.

<AppOnly>

```tsx filename="app/layout.tsx" switcher
import Script from 'next/script'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
      <Script
        src="https://example.com/script.js"
        strategy="beforeInteractive"
      />
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
import Script from 'next/script'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
      <Script
        src="https://example.com/script.js"
        strategy="beforeInteractive"
      />
    </html>
  )
}
```

</AppOnly>

<PagesOnly>

```jsx
import { Html, Head, Main, NextScript } from 'next/document'
import Script from 'next/script'

export default function Document() {
  return (
    <Html>
      <Head />
      <body>
        <Main />
        <NextScript />
        <Script
          src="https://example.com/script.js"
          strategy="beforeInteractive"
        />
      </body>
    </Html>
  )
}
```

</PagesOnly>

> **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.

```jsx filename="app/page.js"
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script src="https://example.com/script.js" strategy="afterInteractive" />
    </>
  )
}
```

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.

```jsx filename="app/page.js"
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script src="https://example.com/script.js" strategy="lazyOnload" />
    </>
  )
}
```

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`](/docs/app/building-your-application/routing/defining-routes) 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 filename="next.config.js"
module.exports = {
  experimental: {
    nextScriptWorkers: true,
  },
}
```

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

```tsx filename="pages/home.tsx" switcher
import Script from 'next/script'

export default function Home() {
```

  return (
    <>
      <Script src="https://example.com/script.js" strategy="worker" />
    </>
)
}

```jsx filename="pages/home.js" switcher
import Script from 'next/script'

export default function Home() {
  return (
    <>
      <Script src="https://example.com/script.js" strategy="worker" />
    </>
  )
}
```

### `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.

```tsx filename="app/page.tsx" switcher
'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.20/lodash.min.js"
        onLoad={() => {
          console.log(_.sample([1, 2, 3, 4]))
        }}
      />
    </>
  )
}
```

```jsx filename="app/page.js" switcher
'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.20/lodash.min.js"
        onLoad={() => {
          console.log(_.sample([1, 2, 3, 4]))
        }}
      />
    </>
  )
}
```

### `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:

<AppOnly>

```tsx filename="app/page.tsx" switcher
'use client'

import { useRef } from 'react'
import Script from 'next/script'
```

~~~
export default function Page() {
  const mapRef = useRef()

  return (
    <>
      <div ref={mapRef}></div>
      <Script
        id="google-maps"
        src="https://maps.googleapis.com/maps/api/js"
        onReady={() => {
          new google.maps.Map(mapRef.current, {
            center: { lat: -34.397, lng: 150.644 },
            zoom: 8,
          })
        }}
      />
    </>
  )
}
```

```jsx filename="app/page.js" switcher
'use client'

import { useRef } from 'react'
import Script from 'next/script'

export default function Page() {
  const mapRef = useRef()

  return (
    <>
      <div ref={mapRef}></div>
      <Script
        id="google-maps"
        src="https://maps.googleapis.com/maps/api/js"
        onReady={() => {
          new google.maps.Map(mapRef.current, {
            center: { lat: -34.397, lng: 150.644 },
            zoom: 8,
          })
        }}
      />
    </>
  )
}
~~~

</AppOnly>

<PagesOnly>

```jsx
import { useRef } from 'react';
import Script from 'next/script';

export default function Page() {
  const mapRef = useRef();

  return (
    <PagesOnly>
      <div ref={mapRef}></div>
      <Script
        id="google-maps"
        src="https://maps.googleapis.com/maps/api/js"
        onReady={() => {
          new google.maps.Map(mapRef.current, {
            center: { lat: -34.397, lng: 150.644 },
            zoom: 8,
          });
        }}
      />
    </>
  );
}
```

</PagesOnly>

### `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:

<AppOnly>

```tsx filename="app/page.tsx" switcher
'use client'
```

~~~~
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onError={(e: Error) => {
          console.error('Script failed to load', e)
        }}
      />
    </>
  )
}
```

```jsx filename="app/page.js" switcher
'use client'

import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onError={(e: Error) => {
          console.error('Script failed to load', e)
        }}
      />
    </>
  )
}
```

</AppOnly>

<PagesOnly>

```jsx
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
~~~~

~~~~
        src="https://example.com/script.js"
        onError={(e: Error) => {
          console.error('Script failed to load', e)
        }}
      />
    </>
  )
}
```

</PagesOnly>

## Version History

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

---
title: <Link>
description: Enable fast client-side navigation with the built-in `next/link`
component.
---

{/* The content of this doc is shared between the app and pages router. You
can use the `<PagesOnly>Content</PagesOnly>` component to add content
that is specific to the Pages Router. Any shared content should not be wrapped
in a component. */}

<PagesOnly>

<details>
  <summary>Examples</summary>

- [Hello World](https://github.com/vercel/next.js/tree/canary/examples/hello-
world)
- [Active className on Link](https://github.com/vercel/next.js/tree/canary/
examples/active-class-name)

</details>
~~~~

</PagesOnly>

`<Link>` is a React component that extends the HTML `<a>` element to provide [prefetching](/docs/app/building-your-application/routing/linking-and-navigating#1-prefetching) and client-side navigation between routes. It is the primary way to navigate between routes in Next.js.

<AppOnly>

```tsx filename="app/page.tsx" switcher
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

```jsx filename="app/page.js" switcher
import Link from 'next/link'

export default function Page() {
  return <Link href="/dashboard">Dashboard</Link>
}
```

</AppOnly>

<PagesOnly>

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

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

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

```jsx
import Link from 'next/link'

function Home() {
  return (
    <ul>
      <li>
        <Link href="/">Home</Link>
      </li>
```

~~~
      <li>
        <Link href="/about">About Us</Link>
      </li>
      <li>
        <Link href="/blog/hello-world">Blog Post</Link>
      </li>
    </ul>
  )
}

export default Home
```

</PagesOnly>

## Props

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

| Prop                     | Example             | Type             | Required |
| ------------------------ | ------------------- | ---------------- | -------- |
| [`href`](#href-required) | `href="/dashboard"` | String or Object | Yes      |
| [`replace`](#replace)    | `replace={false}`   | Boolean          | -        |
| [`scroll`](#scroll)      | `scroll={false}`    | Boolean          | -        |
| [`prefetch`](#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.

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

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

```jsx
// Navigate to /about?name=test
<Link
  href={{
    pathname: '/about',
    query: { name: 'test' },
  }}
~~~

>
  About
</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](https://developer.mozilla.org/docs/Web/API/History_API) stack.

```tsx filename="app/page.tsx" switcher
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/dashboard" replace>
      Dashboard
    </Link>
  )
}
```

```jsx filename="app/page.js" switcher
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/dashboard" replace>
      Dashboard
    </Link>
  )
}
```

### `scroll`

Defaults to `true`. The default behavior of `<Link>` is to scroll to the top of a new route or to maintain the scroll position for backwards and forwards navigation. When `false`, `next/link` will _not_ scroll to the top of the page after a navigation.

```tsx filename="app/page.tsx" switcher
import Link from 'next/link'

export default function Page() {
  return (
```

    <Link href="/dashboard" scroll={false}>
      Dashboard
    </Link>
)
}

```jsx filename="app/page.js" switcher
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/dashboard" scroll={false}>
      Dashboard
    </Link>
  )
}
```

### `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.

```tsx filename="app/page.tsx" switcher
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/dashboard" prefetch={false}>
      Dashboard
    </Link>
  )
}
```

```jsx filename="app/page.js" switcher
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/dashboard" prefetch={false}>
```

```
      Dashboard
    </Link>
  )
}
```

<PagesOnly>

## Other Props

### `legacyBehavior`

An `<a>` element is no longer required as a child of `<Link>`. Add the `legacyBehavior` prop to use the legacy behavior or remove the `<a>` to upgrade. A [codemod is available](/docs/app/building-your-application/upgrading/codemods#new-link) to automatically upgrade your code.

> **Good to know**: when `legacyBehavior` is not set to `true`, all [`anchor`](https://developer.mozilla.org/docs/Web/HTML/Element/a) tag properties can be passed to `next/link` as well such as, `className`, `onClick`, etc.

### `passHref`

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

### `scroll`

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

### `shallow`

Update the path of the current page without rerunning [`getStaticProps`](/docs/pages/building-your-application/data-fetching/get-static-props), [`getServerSideProps`](/docs/pages/building-your-application/data-fetching/get-server-side-props) or [`getInitialProps`](/docs/pages/api-reference/functions/get-initial-props). Defaults to `false`

### `locale`

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

</PagesOnly>

## Examples

### Linking to Dynamic Routes

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

<PagesOnly>

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

```jsx filename="pages/blog/index.js"
import Link from 'next/link'

function Posts({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blog/${post.slug}`}>{post.title}</Link>
        </li>
      ))}
    </ul>
  )
}

export default Posts
```

</PagesOnly>

<AppOnly>

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

```jsx filename="app/blog/page.js"
import Link from 'next/link'

function Page({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>
          <Link href={`/blog/${post.slug}`}>{post.title}</Link>
        </li>
      ))}
```

    </ul>
)
}
~~~

</AppOnly>

<PagesOnly>

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

<AppOnly>

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

</AppOnly>

<PagesOnly>

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

</PagesOnly>

```jsx
import Link from 'next/link'
import styled from 'styled-components'

// This creates a custom component that wraps an <a> tag
const RedLink = styled.a`
  color: red;
`

function NavLink({ href, name }) {
  return (
    <Link href={href} passHref legacyBehavior>
~~~

```
      <RedLink>{name}</RedLink>
    </Link>
  )
}

export default NavLink
```

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

### If the child is a functional component

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

```jsx
import Link from 'next/link'

// `onClick`, `href`, and `ref` need to be passed to the DOM element
// for proper handling
const MyButton = React.forwardRef(({ onClick, href }, ref) => {
  return (
    <a href={href} onClick={onClick} ref={ref}>
      Click Me
    </a>
  )
})

function Home() {
  return (
    <Link href="/about" passHref legacyBehavior>
      <MyButton />
    </Link>
  )
}

export default Home
```

### With URL Object

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

```jsx
import Link from 'next/link'

function Home() {
  return (
    <ul>
      <li>
        <Link
          href={{
            pathname: '/about',
            query: { name: 'test' },
          }}
        >
          About us
        </Link>
      </li>
      <li>
        <Link
          href={{
            pathname: '/blog/[slug]',
            query: { slug: 'my-post' },
          }}
        >
          Blog Post
        </Link>
      </li>
    </ul>
  )
}

export default Home
```

The above example has a link to:

- A predefined route: `/about?name=test`
- A [dynamic route](/docs/app/building-your-application/routing/dynamic-routes): `/blog/my-post`

You can use every property as defined in the [Node.js URL module documentation](https://nodejs.org/api/url.html#url_url_strings_and_url_objects).

### Replace the URL instead of push

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

```jsx
<Link href="/about" replace>
  About us
</Link>
```

### Disable scrolling to the top of the page

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

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

</PagesOnly>

### Middleware

It's common to use [Middleware](/docs/app/building-your-application/routing/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 want to serve a `/dashboard` route that has authenticated and visitor views, you may add something similar to the following in your Middleware to redirect the user to the correct page:

```js filename="middleware.js"
export function middleware(req) {
  const nextUrl = req.nextUrl
  if (nextUrl.pathname === '/dashboard') {
    if (req.cookies.authToken) {
      return NextResponse.rewrite(new URL('/auth/dashboard', req.url))
    } else {
      return NextResponse.rewrite(new URL('/public/dashboard', req.url))
    }
  }
}
```

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

```js
import Link from 'next/link'
import useIsAuthed from './hooks/useIsAuthed'

export default function Page() {
  const isAuthed = useIsAuthed()
  const path = isAuthed ? '/auth/dashboard' : '/dashboard'
  return (
    <Link as="/dashboard" href={path}>
      Dashboard
    </Link>
  )
}
```

<PagesOnly>

> **Good to know**: If you're using [Dynamic Routes](/docs/app/building-your-application/routing/dynamic-routes), you'll need to adapt your `as` and `href` props. For example, if you have a Dynamic Route like `/dashboard/[user]` that you want to present differently via middleware, you would write: `<Link href={{ pathname: '/dashboard/authed/[user]', query: { user: username } }} as="/dashboard/[user]">Profile</Link>`.

</PagesOnly>

## Version History

Version	Changes
`v13.0.0`	No longer requires a child `<a>` tag. A [codemod](/docs/app/building-your-application/upgrading/codemods#remove-a-tags-from-link-components) is provided to automatically update your codebase.
`v10.0.0`	`href` props pointing to a dynamic route are automatically resolved and no longer require an `as` prop.
`v8.0.0`	Improved prefetching performance.

| `v1.0.0` | `next/link` introduced.
|

---
title: <Image>
description: Optimize Images in your Next.js Application using the built-in `next/image` Component.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<details>
  <summary>Examples</summary>

- [Image Component](https://github.com/vercel/next.js/tree/canary/examples/image-component)

</details>

<PagesOnly>

> **Good to know**: If you are using a version of Next.js prior to 13, you'll want to use the [next/legacy/image](/docs/pages/api-reference/components/image-legacy) documentation since the component was renamed.

</PagesOnly>

This API reference will help you understand how to use [props](#props) and [configuration options](#configuration-options) available for the Image Component. For features and usage, please see the [Image Component](/docs/app/building-your-application/optimizing/images) page.

```jsx filename="app/page.js"
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="/profile.png"
      width={500}
      height={500}
      alt="Picture of the author"
    />
  )
}
```

## Props

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

<div style={{ overflowX: 'auto', width: '100%' }}>

Prop	Example	Type	Status
[`src`](#src)	`src="/profile.png"`	String	Required
[`width`](#width)	`width={500}`	Integer (px)	Required
[`height`](#height)	`height={500}`	Integer (px)	Required
[`alt`](#alt)	`alt="Picture of the author"`	String	Required
[`loader`](#loader)	`loader={imageLoader}`	Function	-
[`fill`](#fill)	`fill={true}`	Boolean	-
[`sizes`](#sizes)	`sizes="(max-width: 768px) 100vw, 33vw"`	String	-
[`quality`](#quality)	`quality={80}`	Integer (1-100)	-
[`priority`](#priority)	`priority={true}`	Boolean	-
[`placeholder`](#placeholder)	`placeholder="blur"`	String	-
[`style`](#style)	`style={{objectFit: "contain"}}`	Object	-
[`onLoadingComplete`](#onloadingcomplete)	`onLoadingComplete={img => done())}`	Function	Deprecated
[`onLoad`](#onload)	`onLoad={event => done())}`	Function	-
[`onError`](#onerror)	`onError(event => fail()}`	Function	-
[`loading`](#loading)	`loading="lazy"`	String	-
[`blurDataURL`](#blurdataurl)	`blurDataURL="data:image/jpeg..."`	String	-

</div>

## Required Props

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

```jsx filename="app/page.js"
import Image from 'next/image'

export default function Page() {
  return (
    <div>
      <Image
        src="/profile.png"
        width={500}
        height={500}
        alt="Picture of the author"
      />
    </div>
  )
}
```

### `src`

Must be one of the following:

- A [statically imported](/docs/app/building-your-application/optimizing/images#local-images) image file
- A path string. This can be either an absolute external URL, or an internal path depending on the [loader](#loader) prop.

When using an external URL, you must add it to [remotePatterns](#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](/docs/app/building-your-application/optimizing/images#local-images) or images with the [`fill` property](#fill).

### `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](/docs/app/building-your-application/optimizing/images#local-images) or images with the [`fill` property](#fill).

### `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](https://html.spec.whatwg.org/multipage/images.html#general-guidelines). 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](https://html.spec.whatwg.org/multipage/images.html#a-purely-decorative-image-that-doesn't-add-any-information) or [not intended for the user](https://html.spec.whatwg.org/multipage/images.html#an-image-not-intended-for-the-user), the `alt` property should be an empty string (`alt=""`).

[Learn more](https://html.spec.whatwg.org/multipage/images.html#alt)

## 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](#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`](#src)
- [`width`](#width)
- [`quality`](#quality)

Here is an example of using a custom loader:

```js
'use client'
```

```
import Image from 'next/image'

const imageLoader = ({ src, width, quality }) => {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}

export default function Page() {
  return (
    <Image
      loader={imageLoader}
      src="me.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

<AppOnly>

> **Good to know**: Using props like `loader`, which accept a function, require using [Client Components](/docs/app/building-your-application/rendering/client-components) to serialize the provided function.

</AppOnly>

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

### `fill`

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

A boolean that causes the image to fill the parent element, which is useful when the [`width`](#width) and [`height`](#height) are unknown.

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.

If no styles are applied to the image, the image will stretch 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`](https://developer.mozilla.org/docs/Web/CSS/position)
- [`object-fit`](https://developer.mozilla.org/docs/Web/CSS/object-fit)
- [`object-position`](https://developer.mozilla.org/docs/Web/CSS/object-position)

### `sizes`

A string, similar to a media query, 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`](#fill) or which are [styled to have a responsive size](#responsive-images).

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 `srcset`. 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 changes the behavior of the automatically generated `srcset` value. If no `sizes` value is present, a small `srcset` is generated, suitable for a fixed-size image (1x/2x/etc). If `sizes` is defined, a large `srcset` is generated, suitable for a responsive image (640w/750w/etc). If the `sizes` property includes sizes such as `50vw`, which represent a percentage of the viewport width, then the `srcset` 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:

```jsx
import Image from 'next/image'
```

```
export default function Page() {
  return (
    <div className="grid-element">
      <Image
        fill
        src="/example.png"
        sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw, 33vw"
      />
    </div>
  )
}
```

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](https://web.dev/learn/design/responsive-images/#sizes)
- [mdn](https://developer.mozilla.org/docs/Web/HTML/Element/img#attr-sizes)

### `quality`

```js
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`

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

When *true*, the image will be considered high priority and [preload](https://web.dev/preload-responsive-images/). 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)](https://nextjs.org/learn/seo/web-performance/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`

```js
placeholder = 'empty' // "empty" | "blur" | "data:image/..."
```

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

When `blur`, the [`blurDataURL`](#blurdataurl) property will be used as the placeholder. If `src` is an object from a [static import](/docs/app/building-your-application/optimizing/images#local-images) and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated, except when the image is detected to be animated.

For dynamic images, you must provide the [`blurDataURL`](#blurdataurl) property. Solutions such as [Plaiceholder](https://github.com/joe-bell/plaiceholder) can help with `base64` generation.

When `data:image/...`, the [Data URL](https://developer.mozilla.org/docs/Web/HTTP/Basics_of_HTTP/Data_URIs) will be used as the placeholder while the image is loading.

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

Try it out:

- [Demo the `blur` placeholder](https://image-component.nextjs.gallery/placeholder)
- [Demo the shimmer effect with data URL `placeholder` prop](https://image-component.nextjs.gallery/shimmer)
- [Demo the color effect with `blurDataURL` prop](https://image-component.nextjs.gallery/color)

## 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.

```jsx filename="components/ProfileImage.js"
const imageStyle = {
  borderRadius: '50%',
  border: '1px solid #fff',
}

export default function ProfileImage() {
  return <Image src="..." style={imageStyle} />
}
```

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`

```jsx
'use client'

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

> **Warning**: Deprecated since Next.js *14* in favor of [`onLoad`](#onload).

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

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

<AppOnly>

> **Good to know**: Using props like `onLoadingComplete`, which accept a function, require using [Client Components](/docs/app/building-your-application/rendering/client-components) to serialize the provided function.

</AppOnly>

### `onLoad`

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

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

The callback function will be called with one argument, the Event which has a `target` that references the underlying `<img>` element.

<AppOnly>

> **Good to know**: Using props like `onLoad`, which accept a function, require using [Client Components](/docs/app/building-your-application/rendering/client-components) to serialize the provided function.

</AppOnly>

### `onError`

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

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

<AppOnly>

> **Good to know**: Using props like `onError`, which accept a function, require using [Client Components](/docs/app/building-your-application/rendering/client-components) to serialize the provided function.

</AppOnly>

### `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`](#priority) property instead, which will eagerly preload the image.

```js
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](https://developer.mozilla.org/docs/Web/HTML/Element/img#attr-loading).

### `blurDataURL`

A [Data URL](https://developer.mozilla.org/docs/Web/HTTP/Basics_of_HTTP/Data_URIs) to
be used as a placeholder image before the `src` image successfully loads. Only takes effect when combined
with [`placeholder="blur"`](#placeholder).

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](https://image-component.nextjs.gallery/placeholder)
- [Demo the color effect with `blurDataURL` prop](https://image-component.nextjs.gallery/color)

You can also [generate a solid color Data URL](https://png-pixel.com) to match the image.

### `unoptimized`

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

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

```js
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  return <Image {...props} unoptimized />
}
```

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

```js filename="next.config.js"
module.exports = {
  images: {
    unoptimized: true,
  },
}
```

### 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](#devicesizes) 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:

```js filename="next.config.js"
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'example.com',
        port: '',
        pathname: '/account123/**',
      },
    ],
  },
}
```

> **Good to know**: 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 filename="next.config.js"
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
      },
    ],
  },
}
```

> **Good to know**: 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**: Deprecated since Next.js *14* in favor of strict [`remotePatterns`](#remotepatterns) 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`](#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 filename="next.config.js"
module.exports = {
  images: {
    domains: ['assets.acme.com'],
  },
}
```

### `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 filename="next.config.js"
module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
  },
}
```

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:

```js
'use client'

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

Alternatively, you can use the [`loader` prop](#loader) to configure each instance of `next/image`.

Examples:

- [Custom Image Loader Configuration](/docs/app/api-reference/next-config-js/images#example-loader-configuration)

<AppOnly>

> **Good to know**: Customizing the image loader file, which accepts a function, require using [Client Components](/docs/app/building-your-application/rendering/client-components) to serialize the provided function.

</AppOnly>

## 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`](#sizes) prop to ensure the correct image is served for user's device.

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

```js filename="next.config.js"
module.exports = {
  images: {
    deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
  },
}
```

### `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](#devicesizes) to form the full array of sizes used to generate image [srcset](https://developer.mozilla.org/docs/Web/API/HTMLImageElement/srcset)s.

The reason there are two separate lists is that imageSizes is only used for images which provide a [`sizes`](#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 filename="next.config.js"
module.exports = {
```

  images: {
    imageSizes: [16, 32, 48, 64, 96, 128, 256, 384],
  },
}

### `formats`

The default [Image Optimization API](#loader) 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](#animated-images)), the Image Optimization API will fallback to the original image's format.

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

```js filename="next.config.js"
module.exports = {
  images: {
    formats: ['image/webp'],
  },
}
```

You can enable AVIF support with the following configuration.

```js filename="next.config.js"
module.exports = {
  images: {
    formats: ['image/avif', 'image/webp'],
  },
}
```

> **Good to know**:
>
> - 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.
> - 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](#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`](#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`](#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`](#devicesizes) and [`imageSizes`](#imagesizes) to reduce the total number of possible generated images.
- You can configure [formats](#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](/docs/app/building-your-application/optimizing/images#local-images) which will automatically hash the file contents and cache the image forever with a `Cache-Control` header of `immutable`.

```js filename="next.config.js"
module.exports = {
  images: {
    minimumCacheTTL: 60,
  },
}
```

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`](/docs/app/api-reference/next-config-js/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`](#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 filename="next.config.js"
module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

### `dangerouslyAllowSVG`

The default [loader](#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 a proper [Content Security Policy](/docs/app/building-your-application/configuring/content-security-policy).

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

```js filename="next.config.js"
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
```

    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'; script-src 'none'; sandbox;",
  },
}
~~~

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](#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](#unoptimized) prop.

Responsive Images

The default generated `srcset` contains `1x` and `2x` images in order to support different device pixel ratios. However, you may wish to render a responsive image that stretches with the viewport. In that case, you'll need to set [`sizes`](#sizes) as well as `style` (or `className`).

You can render a responsive image using one of the following methods below.

Responsive image using a static import

If the source image is not dynamic, you can statically import to create a responsive image:

```jsx filename="components/author.js"
import Image from 'next/image'
import me from '../photos/me.jpg'

export default function Author() {
  return (
    <Image
      src={me}
      alt="Picture of the author"
      sizes="100vw"
      style={{
        width: '100%',
        height: 'auto',
      }}
~~~

    />
  )
}

Try it out:

- [Demo the image responsive to viewport](https://image-component.nextjs.gallery/responsive)

### Responsive image with aspect ratio

If the source image is a dynamic or a remote url, you will also need to provide `width` and `height` to set the correct aspect ratio of the responsive image:

```jsx filename="components/page.js"
import Image from 'next/image'

export default function Page({ photoUrl }) {
  return (
    <Image
      src={photoUrl}
      alt="Picture of the author"
      sizes="100vw"
      style={{
        width: '100%',
        height: 'auto',
      }}
      width={500}
      height={300}
    />
  )
}
```

Try it out:

- [Demo the image responsive to viewport](https://image-component.nextjs.gallery/responsive)

### Responsive image with `fill`

If you don't know the aspect ratio, you will need to set the [`fill`](#fill) prop and set `position: relative` on the parent. Optionally, you can set `object-fit` style depending on the desired stretch vs crop behavior:

```jsx filename="app/page.js"
```

import Image from 'next/image'

export default function Page({ photoUrl }) {
  return (
    <div style={{ position: 'relative', width: '500px', height: '300px' }}>
      <Image
        src={photoUrl}
        alt="Picture of the author"
        sizes="500px"
        fill
        style={{
          objectFit: 'contain',
        }}
      />
    </div>
)
}

Try it out:

- [Demo the `fill` prop](https://image-component.nextjs.gallery/fill)

## Theme Detection

If you want to display a different image for light and dark mode, you can create a new component that wraps two `<Image>` components and reveals the correct one based on a CSS media query.

```css filename="components/theme-image.module.css"
.imgDark {
  display: none;
}

@media (prefers-color-scheme: dark) {
  .imgLight {
    display: none;
  }
  .imgDark {
    display: unset;
  }
}
```

```tsx filename="components/theme-image.tsx" switcher
import styles from './theme-image.module.css'
import Image, { ImageProps } from 'next/image'
```

```
type Props = Omit<ImageProps, 'src' | 'priority' | 'loading'> & {
  srcLight: string
  srcDark: string
}

const ThemeImage = (props: Props) => {
  const { srcLight, srcDark, ...rest } = props

  return (
    <>
      <Image {...rest} src={srcLight} className={styles.imgLight} />
      <Image {...rest} src={srcDark} className={styles.imgDark} />
    </>
  )
}
```

```jsx filename="components/theme-image.js" switcher
import styles from './theme-image.module.css'
import Image from 'next/image'

const ThemeImage = (props) => {
  const { srcLight, srcDark, ...rest } = props

  return (
    <>
      <Image {...rest} src={srcLight} className={styles.imgLight} />
      <Image {...rest} src={srcDark} className={styles.imgDark} />
    </>
  )
}
```

> **Good to know**: The default behavior of `loading="lazy"` ensures that only the correct image is loaded. You cannot use `priority` or `loading="eager"` because that would cause both images to load. Instead, you can use [`fetchPriority="high"`](https://developer.mozilla.org/docs/Web/API/HTMLImageElement/fetchPriority).

Try it out:

- [Demo light/dark mode theme detection](https://image-component.nextjs.gallery/theme)

## Known Browser Bugs

This `next/image` component uses browser native [lazy loading](https://caniuse.com/loading-lazy-attr), 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](https://web.dev/cls/) on older browsers before Safari 15 that don't [preserve the aspect ratio](https://caniuse.com/mdn-html_elements_img_aspect_ratio_computed_from_attributes). For more details, see [this MDN video](https://www.youtube.com/watch?v=4-d_SoCHeWE).

- [Safari 15 - 16.3](https://bugs.webkit.org/show_bug.cgi?id=243601) display a gray border while loading. Safari 16.4 [fixed this issue](https://webkit.org/blog/13966/webkit-features-in-safari-16-4/#:~:text=Now%20in%20Safari%2016.4%2C%20a%20gray%20line%20no%20longer%20appears%20to%20mark%20the%20space%20where%20a%20lazy%2Dloaded%20image%20will%20appear%20once%20it%E2%80%99s%20been%20loaded.). Possible solutions:
  - Use CSS `@supports (font: -apple-system-body) and (-webkit-appearance: none) { img[loading="lazy"] { clip-path: inset(0.6px) } }`
  - Use [`priority`](#priority) if the image is above the fold
- [Firefox 67+](https://bugzilla.mozilla.org/show_bug.cgi?id=1556156) displays a white background while loading. Possible solutions:
  - Enable [AVIF `formats`](#formats)
  - Use [`placeholder`](#placeholder)

## Version History

Version	Changes
`v14.0.0`	`onLoadingComplete` prop and `domains` config deprecated.
`v13.4.14`	`placeholder` prop support for `data:/image...`
`v13.2.0`	`contentDispositionType` configuration added.
`v13.0.6`	`ref` prop added.

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

---
title: Font Module
nav_title: Font
description: Optimizing loading web fonts with the built-in `next/font` loaders.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

This API reference will help you understand how to use [`next/font/google`](/docs/app/building-your-application/optimizing/fonts#google-fonts) and [`next/font/local`](/docs/app/building-your-application/optimizing/fonts#local-fonts). For features and usage, please see the [Optimizing Fonts](/docs/app/building-your-application/optimizing/fonts) page.

### Font Function Arguments

For usage, review [Google Fonts](/docs/app/building-your-application/optimizing/fonts#google-fonts) and [Local Fonts](/docs/app/building-your-application/optimizing/fonts#local-fonts).

Key	`font/google`	`font/local`	Type	Required
[`src`](#src)	<Cross size={18} />	<Check size={18} />	String or Array of Objects	Yes
[`weight`](#weight)	<Check size={18} />	<Check size={18} />	String or Array	Required/Optional
[`style`](#style)	<Check size={18} />	<Check size={18} />	String or Array	-
[`subsets`](#subsets)	<Check size={18} />	<Cross size={18} />	Array of Strings	-
[`axes`](#axes)	<Check size={18} />	<Cross size={18} />	Array of Strings	-
[`display`](#display)	<Check size={18} />	<Check size={18} />	String	-
[`preload`](#preload)	<Check size={18} />	<Check size={18} />	Boolean	-
[`fallback`](#fallback)	<Check size={18} />	<Check size={18} />	Array of Strings	-
[`adjustFontFallback`](#adjustfontfallback)	<Check size={18} />	<Check size={18} />	Boolean or String	-
[`variable`](#variable)	<Check size={18} />	<Check size={18} />	String	-
[`declarations`](#declarations)	<Cross size={18} />	<Check size={18} />	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`](https://fonts.google.com/knowledge/glossary/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](https://fonts.google.com/variablefonts) font
- An array of weight values if the font is not a [variable google font](https://fonts.google.com/variablefonts). 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](https://fonts.google.com/variablefonts)

Examples:

- `weight: '400'`: A string for a single weight value - for the font [`Inter`](https://fonts.google.com/specimen/Inter?query=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`](https://developer.mozilla.org/docs/Web/CSS/font-style) with the following possibilities:

- A string [value](https://developer.mozilla.org/docs/Web/CSS/font-style#values) with default value of `'normal'`

- An array of style values if the font is not a [variable google font](https://fonts.google.com/variablefonts). 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](https://developer.mozilla.org/docs/Web/CSS/font-style)
- `style: ['italic','normal']`: An array of 2 values for `next/font/google` - the values are from `normal` and `italic`

### `subsets`

The font [`subsets`](https://fonts.google.com/knowledge/glossary/subsetting) defined by an array of string values with the names of each subset you would like to be [preloaded](/docs/app/building-your-application/optimizing/fonts#specifying-a-subset). Fonts specified via `subsets` will have a link preload tag injected into the head when the [`preload`](#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: ['slnt']`: An array with value `slnt` for the `Inter` variable font which has `slnt` as additional `axes` as shown [here](https://fonts.google.com/variablefonts?vfquery=inter#font-families). You can find the possible `axes` values for your font by using the filter on the [Google variable fonts page](https://fonts.google.com/variablefonts#font-families) and looking for axes other than `wght`

### `display`

The font [`display`](https://developer.mozilla.org/docs/Web/CSS/@font-face/font-display) with possible string [values](https://developer.mozilla.org/docs/Web/CSS/@font-face/font-display#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](/docs/app/building-your-application/optimizing/fonts#preloading) 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](https://web.dev/cls/). 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](https://web.dev/cls/). 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](#css-variables).

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](https://developer.mozilla.org/docs/Web/CSS/@font-face#descriptors) 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`](#classname)
- [`style`](#style-1)
- [CSS Variables](#css-variables)

### `className`

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

```tsx
<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.

```tsx
<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:

```tsx filename="app/page.tsx" switcher
import { Inter } from 'next/font/google'
import styles from '../styles/component.module.css'

const inter = Inter({
  variable: '--font-inter',
})
```

```jsx filename="app/page.js" switcher
import { Inter } from 'next/font/google'
import styles from '../styles/component.module.css'

const inter = Inter({
  variable: '--font-inter',
})
```

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.

```tsx filename="app/page.tsx" switcher
<main className={inter.variable}>
  <p className={styles.text}>Hello World</p>
</main>
```

```jsx filename="app/page.js" switcher
<main className={inter.variable}>
  <p className={styles.text}>Hello World</p>
</main>
```

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

```css filename="styles/component.module.css"
.text {
  font-family: var(--font-inter);
  font-weight: 200;
  font-style: italic;
}
```

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:

```ts filename="styles/fonts.ts" switcher
import { Inter, Lora, Source_Sans_3 } from 'next/font/google'
import localFont from 'next/font/local'

// define your variable fonts
const inter = Inter()
const lora = Lora()
// define 2 weights of a non-variable font
const sourceCodePro400 = Source_Sans_3({ weight: '400' })
const sourceCodePro700 = Source_Sans_3({ weight: '700' })
// define a custom local font where GreatVibes-Regular.ttf is stored in the styles folder
const greatVibes = localFont({ src: './GreatVibes-Regular.ttf' })

export { inter, lora, sourceCodePro400, sourceCodePro700, greatVibes }
```

```js filename="styles/fonts.js" switcher
import { Inter, Lora, Source_Sans_3 } from 'next/font/google'
import localFont from 'next/font/local'

// define your variable fonts
const inter = Inter()
const lora = Lora()
// define 2 weights of a non-variable font
const sourceCodePro400 = Source_Sans_3({ weight: '400' })
const sourceCodePro700 = Source_Sans_3({ weight: '700' })
// define a custom local font where GreatVibes-Regular.ttf is stored in the styles folder
const greatVibes = localFont({ src: './GreatVibes-Regular.ttf' })

export { inter, lora, sourceCodePro400, sourceCodePro700, greatVibes }
```

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

```tsx filename="app/page.tsx" switcher
import { inter, lora, sourceCodePro700, greatVibes } from '../styles/fonts'

export default function Page() {
  return (
    <div>
      <p className={inter.className}>Hello world using Inter font</p>
      <p style={lora.style}>Hello world using Lora font</p>
      <p className={sourceCodePro700.className}>
        Hello world using Source_Sans_3 font with weight 700
      </p>
      <p className={greatVibes.className}>My title in Great Vibes font</p>
    </div>
  )
}
```

```jsx filename="app/page.js" switcher
import { inter, lora, sourceCodePro700, greatVibes } from '../styles/fonts'

export default function Page() {
  return (
    <div>
      <p className={inter.className}>Hello world using Inter font</p>
      <p style={lora.style}>Hello world using Lora font</p>
      <p className={sourceCodePro700.className}>
        Hello world using Source_Sans_3 font with weight 700
      </p>
      <p className={greatVibes.className}>My title in Great Vibes font</p>
    </div>
  )
}
```

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:

```json filename="tsconfig.json"
{
  "compilerOptions": {
    "paths": {
      "@/fonts": ["./styles/fonts"]
    }
  }
}
```

You can now import any font definition as follows:

```tsx filename="app/about/page.tsx" switcher
import { greatVibes, sourceCodePro400 } from '@/fonts'
```

```jsx filename="app/about/page.js" switcher
import { greatVibes, sourceCodePro400 } from '@/fonts'
```

## Version Changes

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

---
title: Components
description: API Reference for Next.js built-in components.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

---
title: favicon, icon, and apple-icon
description: API Reference for the Favicon, Icon and Apple Icon file conventions.

The `favicon`, `icon`, or `apple-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)](#image-files-ico-jpg-png)
- [Using code to generate an icon (.js, .ts, .tsx)](#generate-icons-using-code-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 within your `/app` directory.
The `favicon` image can only be located in the top level of `app/`.

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

File convention	Supported file types	Valid locations
[`favicon`](#favicon)	`.ico`	`app/`
[`icon`](#icon)	`.ico`, `.jpg`, `.jpeg`, `.png`, `.svg`	`app/**/*`
[`apple-icon`](#apple-icon)	`.jpg`, `.jpeg`, `.png`	`app/**/*`

### `favicon`

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

```html filename="<head> output"
<link rel="icon" href="/favicon.ico" sizes="any" />
```

### `icon`

Add an `icon.(ico|jpg|jpeg|png|svg)` image file.

```html filename="<head> output"
<link
  rel="icon"
  href="/icon?<generated>"
  type="image/<generated>"
  sizes="<generated>"
/>
```

### `apple-icon`

Add an `apple-icon.(jpg|jpeg|png)` image file.

```html filename="<head> output"
<link
  rel="apple-touch-icon"
  href="/apple-icon?<generated>"
  type="image/<generated>"
  sizes="<generated>"
/>
```

> **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.
> - Favicons can only be set in the root `/app` segment. If you need more granularity, you can use [`icon`](#icon).
> - 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](https://evilmartians.com/chronicles/how-to-favicon-in-2021-six-files-that-fit-most-needs) where an `.ico` icon is favored over `.svg`.

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

In addition to using [literal image files](#image-files-ico-jpg-png), 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`](/docs/app/api-reference/functions/image-response) API from `next/og`.

```tsx filename="app/icon.tsx" switcher
import { ImageResponse } from 'next/og'

// Route segment config
export const runtime = 'edge'

// Image metadata
export const size = {
  width: 32,
  height: 32,
}
export const contentType = 'image/png'

// Image generation
export default function Icon() {
  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
```

```
        style={{
          fontSize: 24,
          background: 'black',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
          color: 'white',
        }}
      >
        A
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the exported icons size metadata
      // config to also set the ImageResponse's width and height.
      ...size,
    }
  )
}
```

```jsx filename="app/icon.js" switcher
import { ImageResponse } from 'next/og'

// Route segment config
export const runtime = 'edge'

// Image metadata
export const size = {
  width: 32,
  height: 32,
}
export const contentType = 'image/png'

// Image generation
export default function Icon() {
  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
        style={{
          fontSize: 24,
          background: 'black',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
          color: 'white',
        }}
      >
        A
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the exported icons size metadata
      // config to also set the ImageResponse's width and height.
      ...size,
    }
  )
}
```

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

> **Good to know**
>
> - By default, generated icons are [**statically optimized**](/docs/app/building-your-application/rendering/server-components#static-rendering-default) (generated at build time and cached) unless they use [dynamic functions](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions) or uncached data.
> - You can generate multiple icons in the same file using [`generateImageMetadata`](/docs/app/api-reference/functions/generate-image-metadata).
> - You cannot generate a `favicon` icon. Use [`icon`](#icon) or a [favicon.ico](#favicon) file instead.

### Props

The default export function receives the following props:

#### `params` (optional)

An object containing the [dynamic route parameters](/docs/app/building-your-application/routing/dynamic-routes) object from the root segment down to the segment `icon` or `apple-icon` is colocated in.

```tsx filename="app/shop/[slug]/icon.tsx" switcher
export default function Icon({ params }: { params: { slug: string } }) {
  // ...
}
```

```jsx filename="app/shop/[slug]/icon.js" switcher
export default function Icon({ params }) {
  // ...
}
```

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`.

> **Good to know**: `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
[`size`](#size)	`{ width: number; height: number }`
[`contentType`](#contenttype)	`string` - [image MIME type](https://developer.mozilla.org/docs/Web/HTTP/Basics_of_HTTP/MIME_types#image_types)

#### `size`

```tsx filename="icon.tsx | apple-icon.tsx" switcher
export const size = { width: 32, height: 32 }
export default function Icon() {}
```

```jsx filename="icon.js | apple-icon.js" switcher
export const size = { width: 32, height: 32 }

export default function Icon() {}
```

```html filename="<head> output"
<link rel="icon" sizes="32x32" />
```

#### `contentType`

```tsx filename="icon.tsx | apple-icon.tsx" switcher
export const contentType = 'image/png'

export default function Icon() {}
```

```jsx filename="icon.js | apple-icon.js" switcher
export const contentType = 'image/png'

export default function Icon() {}
```

```html filename="<head> output"
<link rel="icon" type="image/png" />
```

#### Route Segment Config

`icon` and `apple-icon` are specialized [Route Handlers](/docs/app/building-your-application/routing/route-handlers) that can use the same [route segment configuration](/docs/app/api-reference/file-conventions/route-segment-config) options as Pages and Layouts.

Option	Type	Default
[`dynamic`](/docs/app/api-reference/file-conventions/route-segment-config#dynamic)	`'auto' | 'force-dynamic' | 'error' | 'force-static'`

`'auto'` |
[`revalidate`](/docs/app/api-reference/file-conventions/route-segment-config#revalidate)	`false | 'force-cache' | 0 | number`	`false`
[`runtime`](/docs/app/api-reference/file-conventions/route-segment-config#runtime)	`'nodejs' | 'edge'`	`'nodejs'`
[`preferredRegion`](/docs/app/api-reference/file-conventions/route-segment-config#preferredregion)	`'auto' | 'global' | 'home' | string | string[]`	`'auto'`

```tsx filename="app/icon.tsx" switcher
export const runtime = 'edge'

export default function Icon() {}
```

```jsx filename="app/icon.js" switcher
export const runtime = 'edge'

export default function Icon() {}
```

## Version History

Version	Changes
`v13.3.0`	`favicon` `icon` and `apple-icon` introduced

---
title: Metadata Files API Reference
nav_title: Metadata Files
description: API documentation for the metadata file conventions.

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.

---

title: manifest.json
description: API Reference for manifest.json file.

Add or generate a `manifest.(json|webmanifest)` file that matches the [Web Manifest Specification](https://developer.mozilla.org/docs/Web/Manifest) in the **root** of `app` directory to provide information about your web application for the browser.

## Static Manifest file

```json filename="app/manifest.json | app/manifest.webmanifest"
{
  "name": "My Next.js Application",
  "short_name": "Next.js App",
  "description": "An application built with Next.js",
  "start_url": "/"
  // ...
}
```

## Generate a Manifest file

Add a `manifest.js` or `manifest.ts` file that returns a [`Manifest` object](#manifest-object).

```ts filename="app/manifest.ts" switcher
import { MetadataRoute } from 'next'

export default function manifest(): MetadataRoute.Manifest {
  return {
    name: 'Next.js App',
    short_name: 'Next.js App',
    description: 'Next.js App',
    start_url: '/',
    display: 'standalone',
    background_color: '#fff',
    theme_color: '#fff',
    icons: [
      {
        src: '/favicon.ico',
        sizes: 'any',
        type: 'image/x-icon',
      },
    ],
  }
}
```

```js filename="app/manifest.js" switcher
export default function manifest() {
  return {
    name: 'Next.js App',
    short_name: 'Next.js App',
    description: 'Next.js App',
    start_url: '/',
    display: 'standalone',
    background_color: '#fff',
    theme_color: '#fff',
    icons: [
      {
        src: '/favicon.ico',
        sizes: 'any',
        type: 'image/x-icon',
      },
    ],
  }
}
```

### Manifest Object

The manifest object contains an extensive list of options that may be updated due to new web standards. For information on all the current options, refer to the `MetadataRoute.Manifest` type in your code editor if using [TypeScript](https://nextjs.org/docs/app/building-your-application/configuring/typescript#typescript-plugin) or see the [MDN](https://developer.mozilla.org/docs/Web/Manifest) docs.

---
title: opengraph-image and twitter-image
description: API Reference for the Open Graph Image and Twitter Image file conventions.

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)](#image-files-jpg-png-gif)

- [Using code to generate images (.js, .ts, .tsx)](#generate-images-using-code-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
[`opengraph-image`](#opengraph-image)	`.jpg`, `.jpeg`, `.png`, `.gif`
[`twitter-image`](#twitter-image)	`.jpg`, `.jpeg`, `.png`, `.gif`
[`opengraph-image.alt`](#opengraph-imagealttxt)	`.txt`
[`twitter-image.alt`](#twitter-imagealttxt)	`.txt`

### `opengraph-image`

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

```html filename="<head> output"
<meta property="og:image" content="<generated>" />
<meta property="og:image:type" content="<generated>" />
<meta property="og:image:width" content="<generated>" />
<meta property="og:image:height" content="<generated>" />
```

### `twitter-image`

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

```html filename="<head> output"
<meta name="twitter:image" content="<generated>" />
<meta name="twitter:image:type" content="<generated>" />
<meta name="twitter:image:width" content="<generated>" />
<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.

```txt filename="opengraph-image.alt.txt"
About Acme
```

```html filename="<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.

```txt filename="twitter-image.alt.txt"
About Acme
```

```html filename="<head> output"
<meta property="twitter:image:alt" content="About Acme" />
```

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

In addition to using [literal image files](#image-files-jpg-png-gif), 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**](/docs/app/building-your-application/rendering/server-components#static-rendering-default) (generated at build time and cached) unless they use [dynamic functions](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions) or uncached data.
> - You can generate multiple Images in the same file using [`generateImageMetadata`](/docs/app/api-reference/functions/generate-image-metadata).

The easiest way to generate an image is to use the [ImageResponse](/docs/app/api-reference/functions/image-response) API from `next/og`.

```tsx filename="app/about/opengraph-image.tsx" switcher
import { ImageResponse } from 'next/og'

// Route segment config
export const runtime = 'edge'

// Image metadata
export const alt = 'About Acme'
export const size = {
  width: 1200,
  height: 630,
}

export const contentType = 'image/png'

// Image generation
export default async function Image() {
  // Font
  const interSemiBold = fetch(
    new URL('./Inter-SemiBold.ttf', import.meta.url)
  ).then((res) => res.arrayBuffer())

  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
        style={{
          fontSize: 128,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        About Acme
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the exported opengraph-image
      // size config to also set the ImageResponse's width and height.
```

~~~
      ...size,
      fonts: [
        {
          name: 'Inter',
          data: await interSemiBold,
          style: 'normal',
          weight: 400,
        },
      ],
    }
  )
}
```

```jsx filename="app/about/opengraph-image.js" switcher
import { ImageResponse } from 'next/og'

// Route segment config
export const runtime = 'edge'

// Image metadata
export const alt = 'About Acme'
export const size = {
  width: 1200,
  height: 630,
}

export const contentType = 'image/png'

// Image generation
export default async function Image() {
  // Font
  const interSemiBold = fetch(
    new URL('./Inter-SemiBold.ttf', import.meta.url)
  ).then((res) => res.arrayBuffer())

  return new ImageResponse(
    (
      // ImageResponse JSX element
      <div
        style={{
          fontSize: 128,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
~~~

```tsx
          justifyContent: 'center',
        }}
      >
        About Acme
      </div>
    ),
    // ImageResponse options
    {
      // For convenience, we can re-use the exported opengraph-image
      // size config to also set the ImageResponse's width and height.
      ...size,
      fonts: [
        {
          name: 'Inter',
          data: await interSemiBold,
          style: 'normal',
          weight: 400,
        },
      ],
    }
  )
}
```

```html filename="<head> output"
<meta property="og:image" content="<generated>" />
<meta property="og:image:alt" content="About Acme" />
<meta property="og:image:type" content="image/png" />
<meta property="og:image:width" content="1200" />
<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](/docs/app/building-your-application/routing/dynamic-routes) object from the root segment down to the segment `opengraph-image` or `twitter-image` is colocated in.

```tsx filename="app/shop/[slug]/opengraph-image.tsx" switcher
export default function Image({ params }: { params: { slug: string } }) {
  // ...
}
```

```jsx filename="app/shop/[slug]/opengraph-image.js" switcher
export default function Image({ params }) {
  // ...
}
```

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`.

> **Good to know**: `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
[`alt`](#alt)	`string`
[`size`](#size)	`{ width: number; height: number }`
[`contentType`](#contenttype)	`string` - [image MIME type](https://developer.mozilla.org/docs/Web/HTTP/Basics_of_HTTP/MIME_types#image_types)

#### `alt`

```tsx filename="opengraph-image.tsx | twitter-image.tsx" switcher
export const alt = 'My images alt text'

export default function Image() {}
```

~~~

```jsx filename="opengraph-image.js | twitter-image.js" switcher
export const alt = 'My images alt text'

export default function Image() {}
~~~

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

#### `size`

```tsx filename="opengraph-image.tsx | twitter-image.tsx" switcher
export const size = { width: 1200, height: 630 }

export default function Image() {}
```

```jsx filename="opengraph-image.js | twitter-image.js" switcher
export const size = { width: 1200, height: 630 }

export default function Image() {}
```

```html filename="<head> output"
<meta property="og:image:width" content="1200" />
<meta property="og:image:height" content="630" />
```

#### `contentType`

```tsx filename="opengraph-image.tsx | twitter-image.tsx" switcher
export const contentType = 'image/png'

export default function Image() {}
```

```jsx filename="opengraph-image.js | twitter-image.js" switcher
export const contentType = 'image/png'

export default function Image() {}
```

```html filename="<head> output"
<meta property="og:image:type" content="image/png" />
```

#### Route Segment Config

`opengraph-image` and `twitter-image` are specialized [Route Handlers](/docs/app/building-your-application/routing/route-handlers) that can use the same [route segment configuration](/docs/app/api-reference/file-conventions/route-segment-config) options as Pages and Layouts.

Option	Type	Default
[`dynamic`](/docs/app/api-reference/file-conventions/route-segment-config#dynamic)	`'auto' | 'force-dynamic' | 'error' | 'force-static'`	`'auto'`
[`revalidate`](/docs/app/api-reference/file-conventions/route-segment-config#revalidate)	`false | 'force-cache' | 0 | number`	`false`
[`runtime`](/docs/app/api-reference/file-conventions/route-segment-config#runtime)	`'nodejs' | 'edge'`	`'nodejs'`
[`preferredRegion`](/docs/app/api-reference/file-conventions/route-segment-config#preferredregion)	`'auto' | 'global' | 'home' | string | string[]`	`'auto'`

```tsx filename="app/opengraph-image.tsx" switcher
export const runtime = 'edge'

export default function Image() {}
```

```jsx filename="app/opengraph-image.js" switcher
export const runtime = 'edge'

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](/docs/app/building-your-application/rendering/server-components#static-rendering-default). You can configure the individual `fetch` [`options`](/docs/app/api-reference/functions/fetch) or route segments [options](/docs/app/api-reference/file-conventions/route-segment-config#revalidate) to change this behavior.

```tsx filename="app/posts/[slug]/opengraph-image.tsx" switcher
import { ImageResponse } from 'next/og'

export const runtime = 'edge'

export const alt = 'About Acme'
export const size = {
  width: 1200,
  height: 630,
}
export const contentType = 'image/png'

export default async function Image({ params }: { params: { slug: string } }) {
  const post = await fetch(`https://.../posts/${params.slug}`).then((res) =>
    res.json()
  )

  return new ImageResponse(
    (
      <div
        style={{
          fontSize: 48,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        {post.title}
      </div>
    ),
    {
      ...size,
    }
  )
}
```

```jsx filename="app/posts/[slug]/opengraph-image.js" switcher
import { ImageResponse } from 'next/og'

export const runtime = 'edge'

export const alt = 'About Acme'
export const size = {
  width: 1200,
  height: 630,
}
export const contentType = 'image/png'

export default async function Image({ params }) {
  const post = await fetch(`https://.../posts/${params.slug}`).then((res) =>
    res.json()
  )

  return new ImageResponse(
    (
      <div
        style={{
          fontSize: 48,
          background: 'white',
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
        }}
      >
        {post.title}
      </div>
    ),
    {
      ...size,
    }
  )
}
```

## Version History

Version	Changes
`v13.3.0`	`opengraph-image` and `twitter-image` introduced.

~~~markdown
---
title: robots.txt
description: API Reference for robots.txt file.
---

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

## Static `robots.txt`

```txt filename="app/robots.txt"
User-Agent: *
Allow: /
Disallow: /private/

Sitemap: https://acme.com/sitemap.xml
```

## Generate a Robots file

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

```ts filename="app/robots.ts" switcher
import { MetadataRoute } from 'next'

export default function robots(): MetadataRoute.Robots {
  return {
    rules: {
      userAgent: '*',
      allow: '/',
      disallow: '/private/',
    },
    sitemap: 'https://acme.com/sitemap.xml',
  }
}
```

```js filename="app/robots.js" switcher
export default function robots() {
  return {
    rules: {
      userAgent: '*',
      allow: '/',
~~~

      disallow: '/private/',
    },
    sitemap: 'https://acme.com/sitemap.xml',
  }
}

Output:

```txt
User-Agent: *
Allow: /
Disallow: /private/

Sitemap: https://acme.com/sitemap.xml
```

### Robots object

```tsx
type Robots = {
  rules:
    | {
        userAgent?: string | string[]
        allow?: string | string[]
        disallow?: string | string[]
        crawlDelay?: number
      }
    | Array<{
        userAgent: string | string[]
        allow?: string | string[]
        disallow?: string | string[]
        crawlDelay?: number
      }>
  sitemap?: string | string[]
  host?: string
}
```

## Version History

Version	Changes
`v13.3.0`	`robots` introduced.

---
title: sitemap.xml

description: API Reference for the sitemap.xml file.

Add or generate a `sitemap.xml` file that matches the [Sitemaps XML format](https://www.sitemaps.org/protocol.html) in the **root** of `app` directory to help search engine crawlers crawl your site more efficiently.

## Static `sitemap.xml`

```xml filename="app/sitemap.xml"
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://acme.com</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>yearly</changefreq>
    <priority>1</priority>
  </url>
  <url>
    <loc>https://acme.com/about</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>https://acme.com/blog</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
</urlset>
```

## Generate a Sitemap

Add a `sitemap.js` or `sitemap.ts` file that returns [`Sitemap`](#sitemap-return-type).

```ts filename="app/sitemap.ts" switcher
import { MetadataRoute } from 'next'

export default function sitemap(): MetadataRoute.Sitemap {
  return [
    {
      url: 'https://acme.com',
      lastModified: new Date(),
      changeFrequency: 'yearly',
      priority: 1,
```

~~~
    },
    {
      url: 'https://acme.com/about',
      lastModified: new Date(),
      changeFrequency: 'monthly',
      priority: 0.8,
    },
    {
      url: 'https://acme.com/blog',
      lastModified: new Date(),
      changeFrequency: 'weekly',
      priority: 0.5,
    },
  ]
}
```

```js filename="app/sitemap.js" switcher
export default function sitemap() {
  return [
    {
      url: 'https://acme.com',
      lastModified: new Date(),
      changeFrequency: 'yearly',
      priority: 1,
    },
    {
      url: 'https://acme.com/about',
      lastModified: new Date(),
      changeFrequency: 'monthly',
      priority: 0.8,
    },
    {
      url: 'https://acme.com/blog',
      lastModified: new Date(),
      changeFrequency: 'weekly',
      priority: 0.5,
    },
  ]
}
```

Output:

```xml filename="acme.com/sitemap.xml"
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
~~~

```
    <loc>https://acme.com</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>yearly</changefreq>
    <priority>1</priority>
  </url>
  <url>
    <loc>https://acme.com/about</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>https://acme.com/blog</loc>
    <lastmod>2023-04-06T15:02:24.021Z</lastmod>
    <changefreq>weekly</changefreq>
    <priority>0.5</priority>
  </url>
</urlset>
```

### Sitemap Return Type

```tsx
type Sitemap = Array<{
  url: string
  lastModified?: string | Date
  changeFrequency?:
    | 'always'
    | 'hourly'
    | 'daily'
    | 'weekly'
    | 'monthly'
    | 'yearly'
    | 'never'
  priority?: number
}>
```

> **Good to know**
>
> - In the future, we will support multiple sitemaps and sitemap indexes.

## Version History

Version	Changes
`v13.3.0`	`sitemap` introduced.

| `v13.4.5` | Add `changeFrequency` and `priority` attributes to sitemaps. |

---
title: default.js
description: API Reference for the default.js file.

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

---
title: error.js
description: API reference for the error.js special file.
related:
  title: Learn more about error handling
  links:
    - app/building-your-application/routing/error-handling

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

It is useful for catching **unexpected** errors that occur in Server Components and Client Components and displaying a fallback UI.

```tsx filename="app/dashboard/error.tsx" switcher
'use client' // Error components must be Client Components

import { useEffect } from 'react'

export default function Error({
  error,
  reset,
}: {
  error: Error & { digest?: string }
  reset: () => void
}) {
  useEffect(() => {
    // Log the error to an error reporting service
    console.error(error)
  }, [error])

  return (
    <div>
      <h2>Something went wrong!</h2>
      <button
        onClick={
          // Attempt to recover by trying to re-render the segment
          () => reset()
```

        }
      >
        Try again
      </button>
    </div>
)
}

```jsx filename="app/dashboard/error.js" switcher
'use client' // Error components must be Client Components

import { useEffect } from 'react'

export default function Error({ error, reset }) {
  useEffect(() => {
    // Log the error to an error reporting service
    console.error(error)
  }, [error])

  return (
    <div>
      <h2>Something went wrong!</h2>
      <button
        onClick={
          // Attempt to recover by trying to re-render the segment
          () => reset()
        }
      >
        Try again
      </button>
    </div>
  )
}
```

## Props

### `error`

An instance of an [`Error`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Error) object forwarded to the `error.js` Client Component.

#### `error.message`

The error message.

- For errors forwarded from Client Components, this will be the original Error's message.
- For errors forwarded from Server Components, this will be a generic error message to avoid leaking sensitive details. `errors.digest` can be used to match the corresponding error in server-side logs.

#### `error.digest`

An automatically generated hash of the error thrown in a Server Component. It can be used to match the corresponding error in server-side logs.

### `reset`

A function to reset the error boundary. 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.

Can be used to prompt the user to attempt to recover from the error.

> **Good to know**:
>
> - `error.js` boundaries must be **[Client Components](/docs/app/building-your-application/rendering/client-components)**.
> - In Production builds, errors forwarded from Server Components will be stripped of specific error details to avoid leaking sensitive information.
> - 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.

```tsx filename="app/global-error.tsx" switcher
'use client'

export default function GlobalError({
  error,
  reset,
}: {
  error: Error & { digest?: string }
```

  reset: () => void
}) {
  return (
    <html>
      <body>
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
      </body>
    </html>
)
}

```jsx filename="app/global-error.js" switcher
'use client'

export default function GlobalError({ error, reset }) {
  return (
    <html>
      <body>
        <h2>Something went wrong!</h2>
        <button onClick={() => reset()}>Try again</button>
      </body>
    </html>
  )
}
```

> **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](https://react.dev/learn/react-developer-tools) to manually toggle Error boundaries.

## not-found.js

The [`not-found`](https://nextjs.org/docs/app/api-reference/file-conventions/not-found) file is used to render UI when the `notFound()` function is thrown within a route segment.

## Version History

Version	Changes
`v13.1.0`	`global-error` introduced.

| `v13.0.0` | `error` introduced. |

---
title: File Conventions
description: API Reference for Next.js Special Files.

---
title: layout.js
description: API reference for the layout.js file.

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

```tsx filename="app/dashboard/layout.tsx" switcher
export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return <section>{children}</section>
}
```

```jsx filename="app/dashboard/layout.js" switcher
export default function DashboardLayout({ children }) {
  return <section>{children}</section>
}
```

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.

```tsx filename="app/layout.tsx" switcher
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

```jsx filename="app/layout.js" switcher
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
    </html>
  )
}
```

## 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](/docs/app/building-your-application/routing/pages-and-layouts#pages) (if it exists) or [Page](/docs/app/building-your-application/routing/pages-and-layouts#pages), but could also be other special files like [Loading](/docs/app/building-your-application/routing/loading-ui-and-streaming) or [Error](/docs/app/building-your-application/routing/error-handling) when applicable.

### `params` (optional)

The [dynamic route parameters](/docs/app/building-your-application/routing/dynamic-routes) object from the root segment down to that layout.

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

For example:

```tsx filename="app/shop/[tag]/[item]/layout.tsx" switcher
export default function ShopLayout({
  children,
  params,
}: {
  children: React.ReactNode
  params: {
    tag: string
    item: string
  }
```

}) {
  // URL -> /shop/shoes/nike-air-max-97
  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
  return <section>{children}</section>
}

```jsx filename="app/shop/[tag]/[item]/layout.js" switcher
export default function ShopLayout({ children, params }) {
  // URL -> /shop/shoes/nike-air-max-97
  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
  return <section>{children}</section>
}
```

## Good to know

### Layouts do not receive `searchParams`

Unlike [Pages](/docs/app/api-reference/file-conventions/page), Layout components **do not** receive the `searchParams` prop. This is because a shared layout is [not re-rendered during navigation](/docs/app/building-your-application/routing/linking-and-navigating#3-partial-rendering) 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`:

<Image
  alt="File structure showing a dashboard folder nesting a layout.tsx file, and settings and analytics folders with their own pages"
  srcLight="/docs/light/shared-dashboard-layout.png"
  srcDark="/docs/dark/shared-dashboard-layout.png"
  width="1600"
  height="687"
/>

When navigating from `/dashboard/settings` to `/dashboard/analytics`, `page.tsx` in `/dashboard/analytics` will rerender on the server, while `dashboard/layout.tsx` will **not** rerender because it's a common UI shared 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`](/docs/app/api-reference/file-conventions/page#searchparams-optional) prop or the [`useSearchParams`](/docs/app/api-reference/functions/use-search-params) 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](/docs/app/api-reference/functions/generate-metadata) which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.
- You can use [route groups](/docs/app/building-your-application/routing/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.

## Version History

Version	Changes
`v13.0.0`	`layout` introduced.

---
title: loading.js
description: API reference for the loading.js file.

A **loading** file can create instant loading states built on [Suspense](/docs/app/building-your-application/routing/loading-ui-and-streaming).

By default, this file is a [Server Component](/docs/app/building-your-application/rendering/server-components) - but can also be used as a Client Component through the `"use client"` directive.

```tsx filename="app/feed/loading.tsx" switcher
export default function Loading() {
```

  // Or a custom loading skeleton component
  return <p>Loading...</p>
}

```jsx filename="app/feed/loading.js" switcher
export default function Loading() {
  // Or a custom loading skeleton component
  return <p>Loading...</p>
}
```

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](https://react.dev/learn/react-developer-tools) to manually toggle Suspense boundaries.

## Version History

Version	Changes
`v13.0.0`	`loading` introduced.

---
title: not-found.js
description: API reference for the not-found.js file.

The **not-found** file is used to render UI when the [`notFound`](/docs/app/api-reference/functions/not-found) function is thrown within a route segment. Along with serving a custom UI, Next.js will return a `200` HTTP status code for streamed responses, and `404` for non-streamed responses.

```tsx filename="app/blog/not-found.tsx" switcher
import Link from 'next/link'

export default function NotFound() {
  return (
    <div>
      <h2>Not Found</h2>
      <p>Could not find requested resource</p>
      <Link href="/">Return Home</Link>
    </div>
  )
```

}

```jsx filename="app/blog/not-found.js" switcher
import Link from 'next/link'

export default function NotFound() {
  return (
    <div>
      <h2>Not Found</h2>
      <p>Could not find requested resource</p>
      <Link href="/">Return Home</Link>
    </div>
  )
}
```

> **Good to know**: 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.

## Data Fetching

By default, `not-found` is a Server Component. You can mark it as `async` to fetch and display data:

```tsx filename="app/blog/not-found.tsx" switcher
import Link from 'next/link'
import { headers } from 'next/headers'

export default async function NotFound() {
  const headersList = headers()
  const domain = headersList.get('host')
  const data = await getSiteData(domain)
  return (
    <div>
      <h2>Not Found: {data.name}</h2>
      <p>Could not find requested resource</p>
      <p>
        View <Link href="/blog">all posts</Link>
      </p>
    </div>
```

)
}

```jsx filename="app/blog/not-found.jsx" switcher
import Link from 'next/link'
import { headers } from 'next/headers'

export default async function NotFound() {
  const headersList = headers()
  const domain = headersList.get('host')
  const data = await getSiteData(domain)
  return (
    <div>
      <h2>Not Found: {data.name}</h2>
      <p>Could not find requested resource</p>
      <p>
        View <Link href="/blog">all posts</Link>
      </p>
    </div>
  )
}
```

## Version History

Version	Changes
`v13.3.0`	Root `app/not-found` handles global unmatched URLs.
`v13.0.0`	`not-found` introduced.

---
title: page.js
description: API reference for the page.js file.

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

```tsx filename="app/blog/[slug]/page.tsx" switcher
export default function Page({
  params,
  searchParams,
}: {
  params: { slug: string }
  searchParams: { [key: string]: string | string[] | undefined }
}) {
  return <h1>My Page</h1>
```

}

```jsx filename="app/blog/[slug]/page.js" switcher
export default function Page({ params, searchParams }) {
  return <h1>My Page</h1>
}
```

## Props

### `params` (optional)

An object containing the [dynamic route parameters](/docs/app/building-your-application/routing/dynamic-routes) 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](https://developer.mozilla.org/docs/Learn/Common_questions/What_is_a_URL#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](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions)** whose values cannot be known ahead of time. Using it will opt the page into **[dynamic rendering](/docs/app/building-your-application/rendering/server-components#dynamic-rendering)** at request time.
> - `searchParams` returns a plain JavaScript object and not a `URLSearchParams` instance.

## Version History

Version	Changes
`v13.0.0`	`page` introduced.
---
title: Route Segment Config
description: Learn about how to configure options for Next.js route segments.

The Route Segment options allows you configure the behavior of a [Page](/docs/app/building-your-application/routing/pages-and-layouts), [Layout](/docs/app/building-your-application/routing/pages-and-layouts), or [Route Handler](/docs/app/building-your-application/routing/route-handlers) by directly exporting the following variables:

Option	Type	Default
[`dynamic`](#dynamic)	`'auto' | 'force-dynamic' | 'error' | 'force-static'`	`'auto'`
[`dynamicParams`](#dynamicparams)	`boolean`	`true`
[`revalidate`](#revalidate)	`false | 'force-cache' | 0 | number`	`false`
[`fetchCache`](#fetchcache)	`'auto' | 'default-cache' | 'only-cache' | 'force-cache' | 'force-no-store' | 'default-no-store' | 'only-no-store'`	`'auto'`
[`runtime`](#runtime)	`'nodejs' | 'edge'`	`'nodejs'`
[`preferredRegion`](#preferredregion)	`'auto' | 'global' | 'home' | string | string[]`	`'auto'`
[`maxDuration`](#maxduration)	`number`	Set by deployment platform

```tsx filename="layout.tsx | page.tsx | route.ts" switcher
export const dynamic = 'auto'
export const dynamicParams = true
export const revalidate = false
export const fetchCache = 'auto'
export const runtime = 'nodejs'
export const preferredRegion = 'auto'
export const maxDuration = 5
```

export default function MyComponent() {}
~~~

```jsx filename="layout.js | page.js | route.js" switcher
export const dynamic = 'auto'
export const dynamicParams = true
export const revalidate = false
export const fetchCache = 'auto'
export const runtime = 'nodejs'
export const preferredRegion = 'auto'
export const maxDuration = 5

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.

```tsx filename="layout.tsx | page.tsx | route.ts" switcher
export const dynamic = 'auto'
// 'auto' | 'force-dynamic' | 'error' | 'force-static'
```

```js filename="layout.js | page.js | route.js" switcher
export const dynamic = 'auto'
// 'auto' | 'force-dynamic' | 'error' | 'force-static'
```

> **Good to know**: 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](/docs/app/building-your-application/rendering/server-components#dynamic-rendering), which will

result in routes being rendered for each user at request time. This option is equivalent to `getServerSideProps()` in the `pages` directory.

- **`'error'`**: Force static rendering and cache the data of a layout or page by causing an error if any components use [dynamic functions](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions) or uncached data. 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`.
  - `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 cache the data of a layout or page by forcing [`cookies()`](/docs/app/api-reference/functions/cookies), [`headers()`](/docs/app/api-reference/functions/headers) and [`useSearchParams()`](/docs/app/api-reference/functions/use-search-params) to return empty values.

> **Good to know**:
>
> - Instructions on [how to migrate](/docs/app/building-your-application/upgrading/app-router-migration#step-6-migrating-data-fetching-methods) from `getServerSideProps` and `getStaticProps` to `dynamic: 'force-dynamic'` and `dynamic: 'error'` can be found in the [upgrade guide](/docs/app/building-your-application/upgrading/app-router-migration#step-6-migrating-data-fetching-methods).

### `dynamicParams`

Control what happens when a dynamic segment is visited that was not generated with [generateStaticParams](/docs/app/api-reference/functions/generate-static-params).

```tsx filename="layout.tsx | page.tsx" switcher
export const dynamicParams = true // true | false,
```

```js filename="layout.js | page.js | route.js" switcher
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](/docs/app/building-your-application/routing/loading-ui-and-streaming#streaming-with-suspense).
> - 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.

```tsx filename="layout.tsx | page.tsx | route.ts" switcher
export const revalidate = false
// false | 'force-cache' | 0 | number
```

```js filename="layout.js | page.js | route.js" switcher
export const revalidate = false
// 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](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions) 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](/docs/app/building-your-application/rendering/server-components#dynamic-rendering) even if no dynamic functions or uncached 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.

> **Good to know**: The `revalidate` option is only available when using the [Node.js Runtime](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes#nodejs-runtime). This means using the `revalidate` option with `runtime = 'edge'` will not work.

#### 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`

<details>
  <summary>This is an advanced option that should only be used if you specifically need to override the default behavior.</summary>

By default, Next.js **will cache** any `fetch()` requests that are reachable **before** any [dynamic functions](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions) are used and **will not cache** `fetch` requests that are discovered **after** dynamic functions are used.

`fetchCache` allows you to override the default `cache` option of all `fetch` requests in a layout or page.

```tsx filename="layout.tsx | page.tsx | route.ts" switcher
export const fetchCache = 'auto'
// 'auto' | 'default-cache' | 'only-cache'
// 'force-cache' | 'force-no-store' | 'default-no-store' | 'only-no-store'
```

```js filename="layout.js | page.js | route.js" switcher
export const fetchCache = 'auto'
// 'auto' | 'default-cache' | 'only-cache'
// 'force-cache' | 'force-no-store' | 'default-no-store' | 'only-no-store'
```

- **`'auto'`** (default)- The default option to cache `fetch` requests before dynamic functions with the `cache` option they provide and not cache `fetch` requests after dynamic functions.
- **`'default-cache'`**: Allow any `cache` option to be passed to `fetch` but if

no option is provided then set the `cache` option to `'force-cache'`. This means that even `fetch` requests after dynamic functions are considered static.
- **`'only-cache'`**: Ensure all `fetch` requests opt into caching by changing the default to `cache: 'force-cache'` if no option is provided and causing an error if any `fetch` requests use `cache: 'no-store'`.
- **`'force-cache'`**: Ensure all `fetch` requests opt into caching by setting the `cache` option of all `fetch` requests to `'force-cache'`.
- **`'default-no-store'`**: Allow any `cache` option to be passed to `fetch` but if no option is provided then set the `cache` option to `'no-store'`. This means that even `fetch` requests before dynamic functions are considered dynamic.
- **`'only-no-store'`**: Ensure all `fetch` requests opt out of caching by changing the default to `cache: 'no-store'` if no option is provided and causing an error if any `fetch` requests use `cache: 'force-cache'`
- **`'force-no-store'`**: Ensure all `fetch` requests opt out of caching by setting the `cache` option of all `fetch` requests to `'no-store'`. This forces all `fetch` requests to be re-fetched every request even if they provide a `'force-cache'` option.

#### Cross-route segment behavior

- Any options set across each layout and page of a single route need to be compatible with each other.
  - If both the `'only-cache'` and `'force-cache'` are provided, then `'force-cache'` wins. If both `'only-no-store'` and `'force-no-store'` are provided, then `'force-no-store'` wins. The force option changes the behavior across the route so a single segment with `'force-*'` would prevent any errors caused by `'only-*'`.
  - The intention of the `'only-*'` and `force-*'` options is to guarantee the whole route is either fully static or fully dynamic. This means:
    - A combination of `'only-cache'` and `'only-no-store'` in a single route is not allowed.
    - A combination of `'force-cache'` and `'force-no-store'` in a single route is not allowed.
  - A parent cannot provide `'default-no-store'` if a child provides `'auto'` or `'*-cache'` since that could make the same fetch have different behavior.
- It is generally recommended to leave shared parent layouts as `'auto'` and customize the options where child segments diverge.

</details>

### `runtime`

```tsx filename="layout.tsx | page.tsx | route.ts" switcher
export const runtime = 'nodejs'
// 'edge' | 'nodejs'
```

```js filename="layout.js | page.js | route.js" switcher
export const runtime = 'nodejs'
// 'edge' | 'nodejs'
```

- **`nodejs`** (default)
- **`edge`**

Learn more about the [Edge and Node.js runtimes](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes).

### `preferredRegion`

```tsx filename="layout.tsx | page.tsx | route.ts" switcher
export const preferredRegion = 'auto'
// 'auto' | 'global' | 'home' | ['iad1', 'sfo1']
```

```js filename="layout.js | page.js | route.js" switcher
export const preferredRegion = 'auto'
// '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.

### `maxDuration`

Based on your deployment platform, you may be able to use a higher default execution time for your function.
This setting allows you to opt into a higher execution time within your plans limit.
Note: This settings requires Next.js `13.4.10` or higher.

```tsx filename="layout.tsx | page.tsx | route.ts" switcher
export const maxDuration = 5
```

```js filename="layout.js | page.js | route.js" switcher
export const maxDuration = 5
```

~~~

> **Good to know**:
>
> - If a `maxDuration` is not specified, the default value is dependent on your deployment platform and plan.

`generateStaticParams`

The `generateStaticParams` function can be used in combination with [dynamic route segments](/docs/app/building-your-application/routing/dynamic-routes) 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](/docs/app/api-reference/functions/generate-static-params) for more details.

---
title: route.js
description: API reference for the route.js special file.

Route Handlers allow you to create custom request handlers for a given route using the Web [Request](https://developer.mozilla.org/docs/Web/API/Request) and [Response](https://developer.mozilla.org/docs/Web/API/Response) APIs.

HTTP Methods

A **route** file allows you to create custom request handlers for a given route. The following [HTTP methods](https://developer.mozilla.org/docs/Web/HTTP/Methods) are supported: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, and `OPTIONS`.

```ts filename="route.ts" switcher
export async function GET(request: Request) {}

export async function HEAD(request: Request) {}

export async function POST(request: Request) {}

export async function PUT(request: Request) {}

export async function DELETE(request: Request) {}

export async function PATCH(request: Request) {}

// If `OPTIONS` is not defined, Next.js will automatically implement `OPTIONS`
~~~

and  set the appropriate Response `Allow` header depending on the other methods defined in the route handler.
export async function OPTIONS(request: Request) {}

```js filename="route.js" switcher
export async function GET(request) {}

export async function HEAD(request) {}

export async function POST(request) {}

export async function PUT(request) {}

export async function DELETE(request) {}

export async function PATCH(request) {}

// If `OPTIONS` is not defined, Next.js will automatically implement `OPTIONS` and  set the appropriate Response `Allow` header depending on the other methods defined in the route handler.
export async function OPTIONS(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](/docs/app/api-reference/functions/next-request) object, which is an extension of the Web [Request](https://developer.mozilla.org/docs/Web/API/Request) API. `NextRequest` gives you further control over the incoming request, including easily accessing `cookies` and an extended, parsed, URL object `nextUrl`.

### `context` (optional)

```ts filename="app/dashboard/[team]/route.js"
export async function GET(request, context: { params }) {
  const team = params.team // '1'
}
```

Currently, the only value of `context` is `params`, which is an object containing the [dynamic route parameters](/docs/app/building-your-application/routing/dynamic-routes) 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](/docs/app/api-reference/functions/next-response).

## Version History

Version	Changes
`v13.2.0`	Route handlers are introduced.

---
title: template.js
description: API Reference for the template.js file.

A **template** file is similar to a [layout](/docs/app/building-your-application/routing/pages-and-layouts#layouts) in that it wraps 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.

```tsx filename="app/template.tsx" switcher
export default function Template({ children }: { children: React.ReactNode }) {
  return <div>{children}</div>
}
```

```jsx filename="app/template.jsx" switcher
export default function Template({ children }) {
  return <div>{children}</div>
}
```

<Image
  alt="template.js special file"
  srcLight="/docs/light/template-special-file.png"

  srcDark="/docs/dark/template-special-file.png"
  width="1600"
  height="444"
/>

While less common, you might choose a template over a layout if you want:

- 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.

## Props

### `children` (required)

Template components should accept and use a `children` prop. `template` is rendered between a [layout](/docs/app/api-reference/file-conventions/layout) and its children. For example:

```jsx filename="Output"
<Layout>
  {/* Note that the template is given a unique key. */}
  <Template key={routeParam}>{children}</Template>
</Layout>
```

> **Good to know**:
>
> - By default, `template` is a [Server Component](/docs/app/building-your-application/rendering/server-components), but can also be used as a [Client Component](/docs/app/building-your-application/rendering/client-components) through the `"use client"` directive.
> - 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.

## Version History

Version	Changes
`v13.0.0`	`template` introduced.

---
title: cookies

description: API Reference for the cookies function.
related:
  title: Next Steps
  description: For more information on what to do next, we recommend the following sections
  links:
    - app/building-your-application/data-fetching/forms-and-mutations
---

The `cookies` function allows you to read the HTTP incoming request cookies from a [Server Component](/docs/app/building-your-application/rendering/server-components) or write outgoing request cookies in a [Server Action](/docs/app/building-your-application/data-fetching/forms-and-mutations) or [Route Handler](/docs/app/building-your-application/routing/route-handlers).

> **Good to know**: `cookies()` is a **[Dynamic Function](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions)** whose returned values cannot be known ahead of time. Using it in a layout or page will opt a route into **[dynamic rendering](/docs/app/building-your-application/rendering/server-components#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.

```jsx filename="app/page.js"
import { cookies } from 'next/headers'

export default function Page() {
  const cookieStore = cookies()
  const theme = cookieStore.get('theme')
  return '...'
}
```

## `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.

```jsx filename="app/page.js"
import { cookies } from 'next/headers'

export default function Page() {
  const cookieStore = cookies()
  return cookieStore.getAll().map((cookie) => (
    <div key={cookie.name}>
      <p>Name: {cookie.name}</p>
      <p>Value: {cookie.value}</p>
    </div>
  ))
}
```

## `cookies().has(name)`

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

```jsx filename="app/page.js"
import { cookies } from 'next/headers'

export default function Page() {
  const cookiesList = cookies()
  const hasCookie = cookiesList.has('theme')
  return '...'
}
```

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

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

> **Good to know**: HTTP does not allow setting cookies after streaming starts, so you must use `.set()` in a [Server Action](/docs/app/building-your-application/data-fetching/forms-and-mutations) or [Route Handler](/docs/app/building-your-application/routing/route-handlers).

```js filename="app/actions.js"
'use server'

import { cookies } from 'next/headers'

async function create(data) {
  cookies().set('name', 'lee')
  // or
  cookies().set('name', 'lee', { secure: true })
  // or
  cookies().set({
    name: 'name',
```

    value: 'lee',
    httpOnly: true,
    path: '/',
  })
}

## Deleting cookies

> **Good to know**: You can only delete cookies in a [Server Action](/docs/app/building-your-application/data-fetching/forms-and-mutations) or [Route Handler](/docs/app/building-your-application/routing/route-handlers).

There are several options for deleting a cookie:

### `cookies().delete(name)`

You can explicitly delete a cookie with a given name.

```js filename="app/actions.js"
'use server'

import { cookies } from 'next/headers'

async function create(data) {
  cookies().delete('name')
}
```

### `cookies().set(name, '')`

Alternatively, you can set a new cookie with the same name and an empty value.

```js filename="app/actions.js"
'use server'

import { cookies } from 'next/headers'

async function create(data) {
  cookies().set('name', '')
}
```

> **Good to know**: `.set()` is only available in a [Server Action](/docs/app/building-your-application/data-fetching/forms-and-mutations) or [Route Handler](/docs/app/building-your-application/routing/route-handlers).

### `cookies().set(name, value, { maxAge: 0 })`

Setting `maxAge` to 0 will immediately expire a cookie.

```js filename="app/actions.js"
'use server'

import { cookies } from 'next/headers'

async function create(data) {
  cookies().set('name', 'value', { maxAge: 0 })
}
```

### `cookies().set(name, value, { expires: timestamp })`

Setting `expires` to any value in the past will immediately expire a cookie.

```js filename="app/actions.js"
'use server'

import { cookies } from 'next/headers'

async function create(data) {
  const oneDay = 24 * 60 * 60 * 1000
  cookies().set('name', 'value', { expires: Date.now() - oneDay })
}
```

> **Good to know**: You can only delete 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 delete.

## Version History

Version	Changes
`v13.0.0`	`cookies` introduced.

---
title: draftMode
description: API Reference for the draftMode function.

The `draftMode` function allows you to detect [Draft Mode](/docs/app/building-your-application/configuring/draft-mode) inside a [Server Component]

(/docs/app/building-your-application/rendering/server-components).

```jsx filename="app/page.js"
import { draftMode } from 'next/headers'

export default function Page() {
  const { isEnabled } = draftMode()
  return (
    <main>
      <h1>My Blog Post</h1>
      <p>Draft Mode is currently {isEnabled ? 'Enabled' : 'Disabled'}</p>
    </main>
  )
}
```

## Version History

Version	Changes
`v13.4.0`	`draftMode` introduced.
---
title: fetch
description: API reference for the extended fetch function.

Next.js extends the native [Web `fetch()` API](https://developer.mozilla.org/docs/Web/API/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.

```tsx filename="app/page.tsx" switcher
export default async function Page() {
  // This request should be cached until manually invalidated.
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be omitted.
  const staticData = await fetch(`https://...`, { cache: 'force-cache' })

  // This request should be refetched on every request.
  // Similar to `getServerSideProps`.
```

  const dynamicData = await fetch(`https://...`, { cache: 'no-store' })

  // This request should be cached with a lifetime of 10 seconds.
  // Similar to `getStaticProps` with the `revalidate` option.
  const revalidatedData = await fetch(`https://...`, {
    next: { revalidate: 10 },
  })

  return <div>...</div>
}

```jsx filename="app/page.js" switcher
export default async function Page() {
  // This request should be cached until manually invalidated.
  // Similar to `getStaticProps`.
  // `force-cache` is the default and can be omitted.
  const staticData = await fetch(`https://...`, { cache: 'force-cache' })

  // This request should be refetched on every request.
  // Similar to `getServerSideProps`.
  const dynamicData = await fetch(`https://...`, { cache: 'no-store' })

  // This request should be cached with a lifetime of 10 seconds.
  // Similar to `getStaticProps` with the `revalidate` option.
  const revalidatedData = await fetch(`https://...`, {
    next: { revalidate: 10 },
  })

  return <div>...</div>
}
```

## `fetch(url, options)`

Since Next.js extends the [Web `fetch()` API](https://developer.mozilla.org/docs/Web/API/Fetch_API), you can use any of the [native options available](https://developer.mozilla.org/docs/Web/API/fetch#parameters).

### `options.cache`

Configure how the request should interact with Next.js [Data Cache](/docs/app/building-your-application/caching#data-cache).

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

- **`force-cache`** (default) - Next.js looks for a matching request in its Data 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](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions) 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`

```ts
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`](/docs/app/api-reference/file-conventions/route-segment-config#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.

### `options.next.tags`

```ts
fetch(`https://...`, { next: { tags: ['collection'] } })
```

Set the cache tags of a resource. Data can then be revalidated on-demand using [`revalidateTag`](https://nextjs.org/docs/app/api-reference/functions/revalidateTag). The max length for a custom tag is 256 characters.

## Version History

Version	Changes
`v13.0.0`	`fetch` introduced.

---
title: generateImageMetadata
description: Learn how to generate multiple images in a single Metadata API special file.
related:
  title: Next Steps
  description: View all the Metadata API options.
  links:
    - app/api-reference/file-conventions/metadata
    - app/building-your-application/optimizing/metadata

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

`generateImageMetadata` function accepts the following parameters:

#### `params` (optional)

An object containing the [dynamic route parameters](/docs/app/building-your-application/routing/dynamic-routes) object from the root segment down to the segment `generateImageMetadata` is called from.

```tsx filename="icon.tsx" switcher
export function generateImageMetadata({
  params,
}: {
  params: { slug: string }
}) {
```

  // ...
}

```jsx filename="icon.js" switcher
export function generateImageMetadata({ params }) {
  // ...
}
```

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 `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
`id`	`string` (required)
`alt`	`string`
`size`	`{ width: number; height: number }`
`contentType`	`string`

```tsx filename="icon.tsx" switcher
import { ImageResponse } from 'next/og'

export function generateImageMetadata() {
  return [
    {
      contentType: 'image/png',
      size: { width: 48, height: 48 },
      id: 'small',
    },
    {
      contentType: 'image/png',
      size: { width: 72, height: 72 },
      id: 'medium',
    },
  ]
```

~~~
}

export default function Icon({ id }: { id: string }) {
  return new ImageResponse(
    (
      <div
        style={{
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
          fontSize: 88,
          background: '#000',
          color: '#fafafa',
        }}
      >
        Icon {id}
      </div>
    )
  )
}
```

```jsx filename="icon.js" switcher
import { ImageResponse } from 'next/og'

export function generateImageMetadata() {
  return [
    {
      contentType: 'image/png',
      size: { width: 48, height: 48 },
      id: 'small',
    },
    {
      contentType: 'image/png',
      size: { width: 72, height: 72 },
      id: 'medium',
    },
  ]
}

export default function Icon({ id }) {
  return new ImageResponse(
    (
      <div
        style={{
~~~

```
          width: '100%',
          height: '100%',
          display: 'flex',
          alignItems: 'center',
          justifyContent: 'center',
          fontSize: 88,
          background: '#000',
          color: '#fafafa',
        }}
      >
        Icon {id}
      </div>
    )
  )
}
```

### Examples

#### Using external data

This example uses the `params` object and external data to generate multiple [Open Graph images](/docs/app/api-reference/file-conventions/metadata/opengraph-image) for a route segment.

```tsx filename="app/products/[id]/opengraph-image.tsx" switcher
import { ImageResponse } from 'next/og'
import { getCaptionForImage, getOGImages } from '@/app/utils/images'

export async function generateImageMetadata({
  params,
}: {
  params: { id: string }
}) {
  const images = await getOGImages(params.id)

  return images.map((image, idx) => ({
    id: idx,
    size: { width: 1200, height: 600 },
    alt: image.text,
    contentType: 'image/png',
  }))
}

export default async function Image({
  params,
  id,
}: {
  params: { id: string }
  id: number
}) {
  const productId = params.id
  const imageId = id
  const text = await getCaptionForImage(productId, imageId)

  return new ImageResponse(
    (
      <div
        style={
          {
            // ...
          }
        }
      >
        {text}
      </div>
    )
  )
}
```

```jsx filename="app/products/[id]/opengraph-image.js" switcher
import { ImageResponse } from 'next/og'
import { getCaptionForImage, getOGImages } from '@/app/utils/images'

export async function generateImageMetadata({ params }) {
  const images = await getOGImages(params.id)

  return images.map((image, idx) => ({
    id: idx,
    size: { width: 1200, height: 600 },
    alt: image.text,
    contentType: 'image/png',
  }))
}

export default async function Image({ params, id }) {
  const productId = params.id
  const imageId = id
  const text = await getCaptionForImage(productId, imageId)

  return new ImageResponse(
    (
      <div
```

~~~~
        style={
          {
            // ...
          }
        }
      >
        {text}
      </div>
    )
  )
}
```

## Version History

| Version   | Changes                            |
| --------- | ---------------------------------- |
| `v13.3.0` | `generateImageMetadata` introduced. |

---
title: Metadata Object and generateMetadata Options
nav_title: generateMetadata
description: Learn how to add Metadata to your Next.js application for improved search engine optimization (SEO) and web shareability.
related:
  title: Next Steps
  description: View all the Metadata API options.
  links:
    - app/api-reference/file-conventions/metadata
    - app/api-reference/functions/generate-viewport
    - app/building-your-application/optimizing/metadata
---

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

```tsx filename="layout.tsx | page.tsx" switcher
import { Metadata } from 'next'

// either Static metadata
export const metadata: Metadata = {
  title: '...',
}

// or Dynamic metadata
export async function generateMetadata({ params }) {
  return {
~~~~

    title: '...',
  }
}
~~~

```jsx filename="layout.js | page.js" switcher
// either Static metadata
export const metadata = {
  title: '...',
}

// or Dynamic metadata
export async function generateMetadata({ params }) {
  return {
    title: '...',
  }
}
~~~

> **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](#metadata-fields) from a `layout.js` or `page.js` file.

```tsx filename="layout.tsx | page.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: '...',
  description: '...',
}

export default function Page() {}
```

```jsx filename="layout.js | page.js" switcher
export const metadata = {
  title: '...',
  description: '...',
}
```

export default function Page() {}

See the [Metadata Fields](#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](#metadata-fields).

```tsx filename="app/products/[id]/page.tsx" switcher
import { Metadata, ResolvingMetadata } from 'next'

type Props = {
  params: { id: string }
  searchParams: { [key: string]: string | string[] | undefined }
}

export async function generateMetadata(
  { params, searchParams }: Props,
  parent: ResolvingMetadata
): Promise<Metadata> {
  // read route params
  const id = params.id

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}

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

```jsx filename="app/products/[id]/page.js" switcher
```

```
export async function generateMetadata({ params, searchParams }, parent) {
  // read route params
  const id = params.id

  // fetch data
  const product = await fetch(`https://.../${id}`).then((res) => res.json())

  // optionally access and extend (rather than replace) parent metadata
  const previousImages = (await parent).openGraph?.images || []

  return {
    title: product.title,
    openGraph: {
      images: ['/some-specific-page-image.jpg', ...previousImages],
    },
  }
}

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

### 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](/docs/app/building-your-application/routing/dynamic-routes) 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](https://developer.mozilla.org/docs/Learn/Common_questions/What_is_a_URL#parameters). 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](#metadata-fields) 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](#the-metadata-object) rather than `generateMetadata`.
> - `fetch` requests are automatically [memoized](/docs/app/building-your-application/caching#request-memoization) for the same data across `generateMetadata`, `generateStaticParams`, Layouts, Pages, and Server Components. React [`cache` can be used](/docs/app/building-your-application/caching#request-memoization) if `fetch` is unavailable.
> - `searchParams` are only available in `page.js` segments.
> - The [`redirect()`](/docs/app/api-reference/functions/redirect) and [`notFound()`](/docs/app/api-reference/functions/not-found) 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](#string) or an optional [template object](#template-object).

#### String

```jsx filename="layout.js | page.js"
export const metadata = {
  title: 'Next.js',
}
```

```html filename="<head> output" hideLineNumbers
<title>Next.js</title>
```

#### Template object

```tsx filename="app/layout.tsx" switcher
import { Metadata } from 'next'
```

export const metadata: Metadata = {
  title: {
    template: '...',
    default: '...',
    absolute: '...',
  },
}

```jsx filename="app/layout.js" switcher
export const metadata = {
  title: {
    default: '...',
    template: '...',
    absolute: '...',
  },
}
```

##### Default

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

```tsx filename="app/layout.tsx"
import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: {
    default: 'Acme',
  },
}
```

```tsx filename="app/about/page.tsx"
import type { Metadata } from 'next'

export const metadata: Metadata = {}

// Output: <title>Acme</title>
```

##### Template

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

```tsx filename="app/layout.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: {
    template: '%s | Acme',
    default: 'Acme', // a default is required when creating a template
  },
}
```

```jsx filename="app/layout.js" switcher
export const metadata = {
  title: {
    template: '%s | Acme',
    default: 'Acme', // a default is required when creating a template
  },
}
```

```tsx filename="app/about/page.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'About',
}

// Output: <title>About | Acme</title>
```

```jsx filename="app/about/page.js" switcher
export const metadata = {
  title: 'About',
}

// 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.

```tsx filename="app/layout.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: {
    template: '%s | Acme',
  },
}
```

```jsx filename="app/layout.js" switcher
export const metadata = {
  title: {
    template: '%s | Acme',
  },
}
```

```tsx filename="app/about/page.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: {
    absolute: 'About',
  },
}

// Output: <title>About</title>
```

```jsx filename="app/about/page.js" switcher
export const metadata = {
  title: {
    absolute: 'About',
  },
}
```

// 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`

```jsx filename="layout.js | page.js"
export const metadata = {
  description: 'The React Framework for the Web',
}
~~~

```html filename="<head> output" hideLineNumbers
<meta name="description" content="The React Framework for the Web" />
```

### Basic Fields

```jsx filename="layout.js | page.js"
export const metadata = {
  generator: 'Next.js',
  applicationName: 'Next.js',
  referrer: 'origin-when-cross-origin',
  keywords: ['Next.js', 'React', 'JavaScript'],
  authors: [{ name: 'Seb' }, { name: 'Josh', url: 'https://nextjs.org' }],
  creator: 'Jiachi Liu',
```

  publisher: 'Sebastian Markbåge',
  formatDetection: {
    email: *false*,
    address: *false*,
    telephone: *false*,
  },
}
~~~

```html filename="<head> output" hideLineNumbers
<meta name="application-name" content="Next.js" />
<meta name="author" content="Seb" />
<link rel="author" href="https://nextjs.org" />
<meta name="author" content="Josh" />
<meta name="generator" content="Next.js" />
<meta name="keywords" content="Next.js,React,JavaScript" />
<meta name="referrer" content="origin-when-cross-origin" />
<meta name="color-scheme" content="dark" />
<meta name="creator" content="Jiachi Liu" />
<meta name="publisher" content="Sebastian Markbåge" />
<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](#default-value).

```jsx filename="layout.js | page.js"
export const metadata = {
  metadataBase: new URL('https://acme.com'),
  alternates: {
    canonical: '/',
    languages: {
      'en-US': '/en-US',
      'de-DE': '/de-DE',
    },
  },
```

  openGraph: {
    images: '/og-image.png',
  },
}

```html filename="<head> output" hideLineNumbers
<link rel="canonical" href="https://acme.com" />
<link rel="alternate" hreflang="en-US" href="https://acme.com/en-US" />
<link rel="alternate" hreflang="de-DE" href="https://acme.com/de-DE" />
<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`](https://vercel.com/docs/concepts/projects/environment-variables/system-environment-variables#:~:text=.-,VERCEL_URL,-The%20domain%20name) 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`:

```tsx filename="app/layout.tsx" switcher
import { Metadata } from 'next'

export const metadata: Metadata = {
  metadataBase: new URL('https://acme.com'),
}
```

```jsx filename="app/layout.js" switcher
export const metadata = {
  metadataBase: new URL('https://acme.com'),
}
```

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

`metadata` field	Resolved URL
`/`	`https://acme.com`
`./`	`https://acme.com`
`payments`	`https://acme.com/payments`
`/payments`	`https://acme.com/payments`
`./payments`	`https://acme.com/payments`
`../payments`	`https://acme.com/payments`
`https://beta.acme.com/payments`	`https://beta.acme.com/payments`

### `openGraph`

```jsx filename="layout.js | page.js"
export const metadata = {
  openGraph: {
    title: 'Next.js',
    description: 'The React Framework for the Web',
    url: 'https://nextjs.org',
    siteName: 'Next.js',
    images: [
      {
        url: 'https://nextjs.org/og.png',
        width: 800,
```

~~~
        height: 600,
      },
      {
        url: 'https://nextjs.org/og-alt.png',
        width: 1800,
        height: 1600,
        alt: 'My custom alt',
      },
    ],
    locale: 'en_US',
    type: 'website',
  },
}
```

```html filename="<head> output" hideLineNumbers
<meta property="og:title" content="Next.js" />
<meta property="og:description" content="The React Framework for the
Web" />
<meta property="og:url" content="https://nextjs.org/" />
<meta property="og:site_name" content="Next.js" />
<meta property="og:locale" content="en_US" />
<meta property="og:image:url" content="https://nextjs.org/og.png" />
<meta property="og:image:width" content="800" />
<meta property="og:image:height" content="600" />
<meta property="og:image:url" content="https://nextjs.org/og-alt.png" />
<meta property="og:image:width" content="1800" />
<meta property="og:image:height" content="1600" />
<meta property="og:image:alt" content="My custom alt" />
<meta property="og:type" content="website" />
```

```jsx filename="layout.js | page.js"
export const metadata = {
  openGraph: {
    title: 'Next.js',
    description: 'The React Framework for the Web',
    type: 'article',
    publishedTime: '2023-01-01T00:00:00.000Z',
    authors: ['Seb', 'Josh'],
  },
}
```

```html filename="<head> output" hideLineNumbers
<meta property="og:title" content="Next.js" />
<meta property="og:description" content="The React Framework for the
~~~

Web" />
<meta property="og:type" content="article" />
<meta property="article:published_time"
content="2023-01-01T00:00:00.000Z" />
<meta property="article:author" content="Seb" />
<meta property="article:author" content="Josh" />

> **Good to know**:
>
> - It may be more convenient to use the [file-based Metadata API](/docs/app/api-reference/file-conventions/metadata/opengraph-image#image-files-jpg-png-gif) 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`

```tsx
import type { Metadata } from 'next'

export const metadata: Metadata = {
  robots: {
    index: false,
    follow: true,
    nocache: true,
    googleBot: {
      index: true,
      follow: false,
      noimageindex: true,
      'max-video-preview': -1,
      'max-image-preview': 'large',
      'max-snippet': -1,
    },
  },
}
```

```html filename="<head> output" hideLineNumbers
<meta name="robots" content="noindex, follow, nocache" />
<meta
  name="googlebot"
  content="index, nofollow, noimageindex, max-video-preview:-1, max-image-preview:large, max-snippet:-1"
/>
```

### `icons`

> **Good to know**: We recommend using the [file-based Metadata API](/docs/app/api-reference/file-conventions/metadata/app-icons#image-files-ico-jpg-png) 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.

```jsx filename="layout.js | page.js"
export const metadata = {
  icons: {
    icon: '/icon.png',
    shortcut: '/shortcut-icon.png',
    apple: '/apple-icon.png',
    other: {
      rel: 'apple-touch-icon-precomposed',
      url: '/apple-touch-icon-precomposed.png',
    },
  },
}
```

```html filename="<head> output" hideLineNumbers
<link rel="shortcut icon" href="/shortcut-icon.png" />
<link rel="icon" href="/icon.png" />
<link rel="apple-touch-icon" href="/apple-icon.png" />
<link
  rel="apple-touch-icon-precomposed"
  href="/apple-touch-icon-precomposed.png"
/>
```

```jsx filename="layout.js | page.js"
export const metadata = {
  icons: {
    icon: [{ url: '/icon.png' }, new URL('/icon.png', 'https://example.com')],
    shortcut: ['/shortcut-icon.png'],
    apple: [
      { url: '/apple-icon.png' },
      { url: '/apple-icon-x3.png', sizes: '180x180', type: 'image/png' },
    ],
    other: [
      {
        rel: 'apple-touch-icon-precomposed',
        url: '/apple-touch-icon-precomposed.png',
      },
```

    ],
  },
}
~~~

```html filename="<head> output" hideLineNumbers
<link rel="shortcut icon" href="/shortcut-icon.png" />
<link rel="icon" href="/icon.png" />
<link rel="apple-touch-icon" href="/apple-icon.png" />
<link
  rel="apple-touch-icon-precomposed"
  href="/apple-touch-icon-precomposed.png"
/>
<link rel="icon" href="https://example.com/icon.png" />
<link
  rel="apple-touch-icon"
  href="/apple-icon-x3.png"
  sizes="180x180"
  type="image/png"
/>
~~~

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

### `themeColor`

> **Deprecated**: The `themeColor` option in `metadata` is deprecated as of Next.js 14. Please use the [`viewport` configuration](/docs/app/api-reference/functions/generate-viewport) instead.

### `manifest`

A web application manifest, as defined in the [Web Application Manifest specification](https://developer.mozilla.org/docs/Web/Manifest).

```jsx filename="layout.js | page.js"
export const metadata = {
  manifest: 'https://nextjs.org/manifest.json',
}
```

```html filename="<head> output" hideLineNumbers
<link rel="manifest" href="https://nextjs.org/manifest.json" />
```

### `twitter`

Learn more about the [Twitter Card markup reference](https://developer.twitter.com/en/docs/twitter-for-websites/cards/overview/markup).

```jsx filename="layout.js | page.js"
export const metadata = {
  twitter: {
    card: 'summary_large_image',
    title: 'Next.js',
    description: 'The React Framework for the Web',
    siteId: '1467726470533754880',
    creator: '@nextjs',
    creatorId: '1467726470533754880',
    images: ['https://nextjs.org/og.png'],
  },
}
```

```html filename="<head> output" hideLineNumbers
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:site:id" content="1467726470533754880" />
<meta name="twitter:creator" content="@nextjs" />
<meta name="twitter:creator:id" content="1467726470533754880" />
<meta name="twitter:title" content="Next.js" />
<meta name="twitter:description" content="The React Framework for the
Web" />
<meta name="twitter:image" content="https://nextjs.org/og.png" />
```

```jsx filename="layout.js | page.js"
export const metadata = {
  twitter: {
    card: 'app',
    title: 'Next.js',
    description: 'The React Framework for the Web',
    siteId: '1467726470533754880',
    creator: '@nextjs',
    creatorId: '1467726470533754880',
    images: {
      url: 'https://nextjs.org/og.png',
      alt: 'Next.js Logo',
    },
    app: {
      name: 'twitter_app',
      id: {
        iphone: 'twitter_app://iphone',
        ipad: 'twitter_app://ipad',
```

        googleplay: 'twitter_app://googleplay',
      },
      url: {
        iphone: 'https://iphone_url',
        ipad: 'https://ipad_url',
      },
    },
  },
}
~~~

```html filename="<head> output" hideLineNumbers
<meta name="twitter:site:id" content="1467726470533754880" />
<meta name="twitter:creator" content="@nextjs" />
<meta name="twitter:creator:id" content="1467726470533754880" />
<meta name="twitter:title" content="Next.js" />
<meta name="twitter:description" content="The React Framework for the Web" />
<meta name="twitter:card" content="app" />
<meta name="twitter:image" content="https://nextjs.org/og.png" />
<meta name="twitter:image:alt" content="Next.js Logo" />
<meta name="twitter:app:name:iphone" content="twitter_app" />
<meta name="twitter:app:id:iphone" content="twitter_app://iphone" />
<meta name="twitter:app:id:ipad" content="twitter_app://ipad" />
<meta name="twitter:app:id:googleplay" content="twitter_app://googleplay" />
<meta name="twitter:app:url:iphone" content="https://iphone_url" />
<meta name="twitter:app:url:ipad" content="https://ipad_url" />
<meta name="twitter:app:name:ipad" content="twitter_app" />
<meta name="twitter:app:name:googleplay" content="twitter_app" />
~~~

### `viewport`

> **Deprecated**: The `themeColor` option in `metadata` is deprecated as of Next.js 14. Please use the [`viewport` configuration](/docs/app/api-reference/functions/generate-viewport) instead.

### `verification`

```jsx filename="layout.js | page.js"
export const metadata = {
  verification: {
    google: 'google',
    yandex: 'yandex',
    yahoo: 'yahoo',
    other: {
      me: ['my-email', 'my-link'],
```

    },
  },
}
~~~

```html filename="<head> output" hideLineNumbers
<meta name="google-site-verification" content="google" />
<meta name="y_key" content="yahoo" />
<meta name="yandex-verification" content="yandex" />
<meta name="me" content="my-email" />
<meta name="me" content="my-link" />
~~~

### `appleWebApp`

```jsx filename="layout.js | page.js"
export const metadata = {
  itunes: {
    appId: 'myAppStoreID',
    appArgument: 'myAppArgument',
  },
  appleWebApp: {
    title: 'Apple Web App',
    statusBarStyle: 'black-translucent',
    startupImage: [
      '/assets/startup/apple-touch-startup-image-768x1004.png',
      {
        url: '/assets/startup/apple-touch-startup-image-1536x2008.png',
        media: '(device-width: 768px) and (device-height: 1024px)',
      },
    ],
  },
}
```

```html filename="<head> output" hideLineNumbers
<meta
  name="apple-itunes-app"
  content="app-id=myAppStoreID, app-argument=myAppArgument"
/>
<meta name="apple-mobile-web-app-capable" content="yes" />
<meta name="apple-mobile-web-app-title" content="Apple Web App" />
<link
  href="/assets/startup/apple-touch-startup-image-768x1004.png"
  rel="apple-touch-startup-image"
/>
<link
```

  href="/assets/startup/apple-touch-startup-image-1536x2008.png"
  media="(device-width: 768px) and (device-height: 1024px)"
  rel="apple-touch-startup-image"
/>
<meta
  name="apple-mobile-web-app-status-bar-style"
  content="black-translucent"
/>
~~~

`alternates`

```jsx filename="layout.js | page.js"
export const metadata = {
  alternates: {
    canonical: 'https://nextjs.org',
    languages: {
      'en-US': 'https://nextjs.org/en-US',
      'de-DE': 'https://nextjs.org/de-DE',
    },
    media: {
      'only screen and (max-width: 600px)': 'https://nextjs.org/mobile',
    },
    types: {
      'application/rss+xml': 'https://nextjs.org/rss',
    },
  },
}
~~~

```html filename="<head> output" hideLineNumbers
<link rel="canonical" href="https://nextjs.org" />
<link rel="alternate" hreflang="en-US" href="https://nextjs.org/en-US" />
<link rel="alternate" hreflang="de-DE" href="https://nextjs.org/de-DE" />
<link
  rel="alternate"
  media="only screen and (max-width: 600px)"
  href="https://nextjs.org/mobile"
/>
<link
  rel="alternate"
  type="application/rss+xml"
  href="https://nextjs.org/rss"
/>
```

### `appLinks`

```jsx filename="layout.js | page.js"
export const metadata = {
  appLinks: {
    ios: {
      url: 'https://nextjs.org/ios',
      app_store_id: 'app_store_id',
    },
    android: {
      package: 'com.example.android/package',
      app_name: 'app_name_android',
    },
    web: {
      url: 'https://nextjs.org/web',
      should_fallback: true,
    },
  },
}
```

```html filename="<head> output" hideLineNumbers
<meta property="al:ios:url" content="https://nextjs.org/ios" />
<meta property="al:ios:app_store_id" content="app_store_id" />
<meta property="al:android:package" content="com.example.android/
package" />
<meta property="al:android:app_name" content="app_name_android" />
<meta property="al:web:url" content="https://nextjs.org/web" />
<meta property="al:web:should_fallback" content="true" />
```

### `archives`

Describes a collection of records, documents, or other materials of historical interest ([source](https://www.w3.org/TR/2011/WD-html5-20110113/links.html#rel-archives)).

```jsx filename="layout.js | page.js"
export const metadata = {
  archives: ['https://nextjs.org/13'],
}
```

```html filename="<head> output" hideLineNumbers
<link rel="archives" href="https://nextjs.org/13" />
```

### `assets`

```jsx filename="layout.js | page.js"
export const metadata = {
  assets: ['https://nextjs.org/assets'],
}
```

```html filename="<head> output" hideLineNumbers
<link rel="assets" href="https://nextjs.org/assets" />
```

### `bookmarks`

```jsx filename="layout.js | page.js"
export const metadata = {
  bookmarks: ['https://nextjs.org/13'],
}
```

```html filename="<head> output" hideLineNumbers
<link rel="bookmarks" href="https://nextjs.org/13" />
```

### `category`

```jsx filename="layout.js | page.js"
export const metadata = {
  category: 'technology',
}
```

```html filename="<head> output" hideLineNumbers
<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.

```jsx filename="layout.js | page.js"
export const metadata = {
  other: {
    custom: 'meta',
  },
```

}
~~~

```html filename="<head> output" hideLineNumbers
<meta name="custom" content="meta" />
~~~

If you want to generate multiple same key meta tags you can use array value.

```jsx filename="layout.js | page.js"
export const metadata = {
  other: {
    custom: ['meta1', 'meta2'],
  },
}
```

```html filename="<head> output" hideLineNumbers
<meta name="custom" content="meta1" /> <meta name="custom"
content="meta2" />
```

## 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()`](/docs/app/api-reference/functions/redirect), [Middleware](/docs/app/building-your-application/routing/middleware#nextresponse), [Security Headers](/docs/app/api-reference/next-config-js/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](/docs/app/building-your-application/styling/css-modules).
`<script>`	Learn more about [using scripts](/docs/app/building-your-application/optimizing/scripts).

|
| `<link rel="stylesheet" />` | `import` stylesheets directly in the layout or page itself.
|
| `<link rel="preload />` | Use [ReactDOM preload method](#link-relpreload)
|
| `<link rel="preconnect" />` | Use [ReactDOM preconnect method](#link-relpreconnect)
|
| `<link rel="dns-prefetch" />` | Use [ReactDOM prefetchDNS method](#link-reldns-prefetch)
|

### Resource hints

The `<link>` element has a number of `rel` keywords that can be used to hint to the browser that an 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](https://github.com/facebook/react/pull/26237) to safely insert them into the `<head>` of the document.

```tsx filename="app/preload-resources.tsx" switcher
'use client'

import ReactDOM from 'react-dom'

export function PreloadResources() {
  ReactDOM.preload('...', { as: '...' })
  ReactDOM.preconnect('...', { crossOrigin: '...' })
  ReactDOM.prefetchDNS('...')

  return null
}
```

```jsx filename="app/preload-resources.js" switcher
'use client'

import ReactDOM from 'react-dom'

export function PreloadResources() {
  ReactDOM.preload('...', { as: '...' })
  ReactDOM.preconnect('...', { crossOrigin: '...' })
  ReactDOM.prefetchDNS('...')
```

  return null
}

##### `<link rel="preload">`

Start loading a resource early in the page rendering (browser) lifecycle. [MDN Docs](https://developer.mozilla.org/docs/Web/HTML/Attributes/rel/preload).

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

```html filename="<head> output" hideLineNumbers
<link rel="preload" href="..." as="..." />
```

##### `<link rel="preconnect">`

Preemptively initiate a connection to an origin. [MDN Docs](https://developer.mozilla.org/docs/Web/HTML/Attributes/rel/preconnect).

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

```html filename="<head> output" hideLineNumbers
<link rel="preconnect" href="..." crossorigin />
```

##### `<link rel="dns-prefetch">`

Attempt to resolve a domain name before resources get requested. [MDN Docs](https://developer.mozilla.org/docs/Web/HTML/Attributes/rel/dns-prefetch).

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

```html filename="<head> output" hideLineNumbers
<link rel="dns-prefetch" href="..." />
```

> **Good to know**:
>

> - These methods are currently only supported in Client Components, which 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](/docs/app/building-your-application/configuring/typescript) in your IDE, you do not need to manually add the type, but you can still explicitly add it if you want.

### `metadata` object

```tsx
import type { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'Next.js',
}
```

### `generateMetadata` function

#### Regular function

```tsx
import type { Metadata } from 'next'

export function generateMetadata(): Metadata {
  return {
    title: 'Next.js',
  }
}
```

#### Async function

```tsx
import type { Metadata } from 'next'

export async function generateMetadata(): Promise<Metadata> {
  return {
    title: 'Next.js',
```

  }
}

#### With segment props

```tsx
import type { Metadata } from 'next'

type Props = {
  params: { id: string }
  searchParams: { [key: string]: string | string[] | undefined }
}

export function generateMetadata({ params, searchParams }: Props): Metadata
{
  return {
    title: 'Next.js',
  }
}

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

#### With parent metadata

```tsx
import type { Metadata, ResolvingMetadata } from 'next'

export async function generateMetadata(
  { params, searchParams }: Props,
  parent: ResolvingMetadata
): Promise<Metadata> {
  return {
    title: 'Next.js',
  }
}
```

#### JavaScript Projects

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

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

}

## Version History

Version	Changes
`v13.2.0`	`viewport`, `themeColor`, and `colorScheme` deprecated in favor of the [`viewport` configuration](/docs/app/api-reference/functions/generate-viewport).
`v13.2.0`	`metadata` and `generateMetadata` introduced.

---
title: generateStaticParams
description: API reference for the generateStaticParams function.

The `generateStaticParams` function can be used in combination with [dynamic route segments](/docs/app/building-your-application/routing/dynamic-routes) to [**statically generate**](/docs/app/building-your-application/rendering/server-components#static-rendering-default) routes at build time instead of on-demand at request time.

```jsx filename="app/blog/[slug]/page.js"
// Return a list of `params` to populate the [slug] dynamic segment
export async function generateStaticParams() {
  const posts = await fetch('https://.../posts').then((res) => res.json())

  return posts.map((post) => ({
    slug: post.slug,
  }))
}

// Multiple versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
export default function Page({ params }) {
  const { slug } = params
  // ...
}
```

> **Good to know**
>

> - You can use the [`dynamicParams`](/docs/app/api-reference/file-conventions/route-segment-config#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`](/docs/pages/api-reference/functions/get-static-paths) 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](#multiple-dynamic-segments-in-a-route).

## 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
`/product/[id]`	`{ id: string }[]`
`/products/[category]/[product]`	`{ category: string, product: string }[]`
`/products/[...slug]`	`{ slug: string[] }[]`

## Single Dynamic Segment

```tsx filename="app/product/[id]/page.tsx" switcher
export function generateStaticParams() {
  return [{ id: '1' }, { id: '2' }, { id: '3' }]
}

// Three versions of this page will be statically generated
```

~~~
// using the `params` returned by `generateStaticParams`
// - /product/1
// - /product/2
// - /product/3
export default function Page({ params }: { params: { id: string } }) {
  const { id } = params
  // ...
}
```

```jsx filename="app/product/[id]/page.js" switcher
export function generateStaticParams() {
  return [{ id: '1' }, { id: '2' }, { id: '3' }]
}

// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /product/1
// - /product/2
// - /product/3
export default function Page({ params }) {
  const { id } = params
  // ...
}
```

## Multiple Dynamic Segments

```tsx filename="app/products/[category]/[product]/page.tsx" switcher
export function generateStaticParams() {
  return [
    { category: 'a', product: '1' },
    { category: 'b', product: '2' },
    { category: 'c', product: '3' },
  ]
}

// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /products/a/1
// - /products/b/2
// - /products/c/3
export default function Page({
  params,
}: {
  params: { category: string; product: string }
}) {
~~~

```
  const { category, product } = params
  // ...
}
```

```jsx filename="app/products/[category]/[product]/page.js" switcher
export function generateStaticParams() {
  return [
    { category: 'a', product: '1' },
    { category: 'b', product: '2' },
    { category: 'c', product: '3' },
  ]
}

// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /products/a/1
// - /products/b/2
// - /products/c/3
export default function Page({ params }) {
  const { category, product } = params
  // ...
}
```

## Catch-all Dynamic Segment

```tsx filename="app/product/[...slug]/page.tsx" switcher
export function generateStaticParams() {
  return [{ slug: ['a', '1'] }, { slug: ['b', '2'] }, { slug: ['c', '3'] }]
}

// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /product/a/1
// - /product/b/2
// - /product/c/3
export default function Page({ params }: { params: { slug: string[] } }) {
  const { slug } = params
  // ...
}
```

```jsx filename="app/product/[...slug]/page.js" switcher
export function generateStaticParams() {
  return [{ slug: ['a', '1'] }, { slug: ['b', '2'] }, { slug: ['c', '3'] }]
}
```

```
// Three versions of this page will be statically generated
// using the `params` returned by `generateStaticParams`
// - /product/a/1
// - /product/b/2
// - /product/c/3
export default function Page({ params }) {
  const { slug } = params
  // ...
}
```

## 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.

```tsx filename="app/products/[category]/[product]/page.tsx" switcher
// Generate segments for both [category] and [product]
export async function generateStaticParams() {
  const products = await fetch('https://.../products').then((res) => res.json())

  return products.map((product) => ({
    category: product.category.slug,
    product: product.id,
  }))
}

export default function Page({
  params,
}: {
  params: { category: string; product: string }
```

}) {
  // ...
}

```jsx filename="app/products/[category]/[product]/page.js" switcher
// Generate segments for both [category] and [product]
export async function generateStaticParams() {
  const products = await fetch('https://.../products').then((res) => res.json())

  return products.map((product) => ({
    category: product.category.slug,
    product: product.id,
  }))
}

export default function Page({ params }) {
  // ...
}
```

### Generate params from the top down

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

```tsx filename="app/products/[category]/layout.tsx" switcher
// Generate segments for [category]
export async function generateStaticParams() {
  const products = await fetch('https://.../products').then((res) => res.json())

  return products.map((product) => ({
    category: product.category.slug,
  }))
}

export default function Layout({ params }: { params: { category: string } }) {
  // ...
}
```

```jsx filename="app/products/[category]/layout.js" switcher
// Generate segments for [category]
export async function generateStaticParams() {
  const products = await fetch('https://.../products').then((res) => res.json())

  return products.map((product) => ({
    category: product.category.slug,
  }))
}

export default function Layout({ params }) {
  // ...
}
```

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.

```tsx filename="app/products/[category]/[product]/page.tsx" switcher
// Generate segments for [product] using the `params` passed from
// the parent segment's `generateStaticParams` function
export async function generateStaticParams({
  params: { category },
}: {
  params: { category: string }
}) {
  const products = await fetch(
    `https://.../products?category=${category}`
  ).then((res) => res.json())

  return products.map((product) => ({
    product: product.id,
  }))
}

export default function Page({
  params,
}: {
  params: { category: string; product: string }
}) {
  // ...
}
```

```jsx filename="app/products/[category]/[product]/page.js" switcher
// Generate segments for [product] using the `params` passed from
// the parent segment's `generateStaticParams` function
export async function generateStaticParams({ params: { category } }) {
  const products = await fetch(
    `https://.../products?category=${category}`
  ).then((res) => res.json())

  return products.map((product) => ({
    product: product.id,
  }))
}

export default function Page({ params }) {
  // ...
}
```

> **Good to know**: `fetch` requests are automatically [memoized](/docs/app/building-your-application/caching#request-memoization) for the same data across all `generate`-prefixed functions, Layouts, Pages, and Server Components. React [`cache` can be used](/docs/app/building-your-application/caching#request-memoization) if `fetch` is unavailable.

## Version History

Version	Changes
`v13.0.0`	`generateStaticParams` introduced.

---
title: generateViewport
description: API Reference for the generateViewport function.
related:
  title: Next Steps
  description: View all the Metadata API options.
  links:
    - app/api-reference/file-conventions/metadata
    - app/building-your-application/optimizing/metadata

You can customize the initial viewport of the page with the static `viewport` object or the dynamic `generateViewport` function.

> **Good to know**:
>
> - The `viewport` object and `generateViewport` function exports are **only supported in Server Components**.
> - You cannot export both the `viewport` object and `generateViewport` function from the same route segment.
> - If you're coming from migrating `metadata` exports, you can use [metadata-to-viewport-export codemod](/docs/app/building-your-application/

upgrading/codemods#metadata-to-viewport-export) to update your changes.

## The `viewport` object

To define the viewport options, export a `viewport` object from a `layout.js` or `page.js` file.

```tsx filename="layout.tsx | page.tsx" switcher
import { Viewport } from 'next'

export const viewport: Viewport = {
  themeColor: 'black',
}

export default function Page() {}
```

```jsx filename="layout.js | page.js" switcher
export const viewport = {
  themeColor: 'black',
}

export default function Page() {}
```

## `generateViewport` function

`generateViewport` should return a [`Viewport` object](#viewport-fields) containing one or more viewport fields.

```tsx filename="layout.tsx | page.tsx" switcher
export function generateViewport({ params }) {
  return {
    themeColor: '...',
  }
}
```

```jsx filename="layout.js | page.js" switcher
export function generateViewport({ params }) {
  return {
    themeColor: '...',
  }
}
```

> **Good to know**:

>
> - If the viewport doesn't depend on runtime information, it should be defined using the static [`viewport` object](#the-viewport-object) rather than `generateMetadata`.

## Viewport Fields

### `themeColor`

Learn more about [theme-color](https://developer.mozilla.org/docs/Web/HTML/Element/meta/name/theme-color).

Simple theme color

```jsx filename="layout.js | page.js"
export const viewport = {
  themeColor: 'black',
}
```

```html filename="<head> output" hideLineNumbers
<meta name="theme-color" content="black" />
```

With media attribute

```jsx filename="layout.js | page.js"
export const viewport = {
  themeColor: [
    { media: '(prefers-color-scheme: light)', color: 'cyan' },
    { media: '(prefers-color-scheme: dark)', color: 'black' },
  ],
}
```

```html filename="<head> output" hideLineNumbers
<meta name="theme-color" media="(prefers-color-scheme: light)"
content="cyan" />
<meta name="theme-color" media="(prefers-color-scheme: dark)"
content="black" />
```

### `width`, `initialScale`, and `maximumScale`

> **Good to know**: 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.

```jsx filename="layout.js | page.js"
export const viewport = {
  width: 'device-width',
  initialScale: 1,
  maximumScale: 1,
  // Also supported by less commonly used
  // interactiveWidget: 'resizes-visual',
}
```

```html filename="<head> output" hideLineNumbers
<meta
  name="viewport"
  content="width=device-width, initial-scale=1, maximum-scale=1"
/>
```

### `colorScheme`

Learn more about [`color-scheme`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta/name#:~:text=color%2Dscheme%3A%20specifies,of%20the%20following%3A).

```jsx filename="layout.js | page.js"
export const viewport = {
  colorScheme: 'dark',
}
```

```html filename="<head> output" hideLineNumbers
<meta name="color-scheme" content="dark" />
```

## Types

You can add type safety to your viewport object by using the `Viewport` type. If you are using the [built-in TypeScript plugin](/docs/app/building-your-application/configuring/typescript) in your IDE, you do not need to manually add the type, but you can still explicitly add it if you want.

### `viewport` object

```tsx
import type { Viewport } from 'next'
```

export const viewport: Viewport = {
  themeColor: 'black',
}

### `generateViewport` function

#### Regular function

```tsx
import type { Viewport } from 'next'

export function generateViewport(): Viewport {
  return {
    themeColor: 'black',
  }
}
```

#### With segment props

```tsx
import type { Viewport } from 'next'

type Props = {
  params: { id: string }
  searchParams: { [key: string]: string | string[] | undefined }
}

export function generateViewport({ params, searchParams }: Props): Viewport {
  return {
    themeColor: 'black',
  }
}

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

#### JavaScript Projects

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

```js
/** @type {import("next").Viewport} */
export const viewport = {
  themeColor: 'black',
```

}

## Version History

Version	Changes
`v14.0.0`	`viewport` and `generateViewport` introduced.

---
title: headers
description: API reference for the headers function.

The `headers` function allows you to read the HTTP incoming request headers from a [Server Component](/docs/app/building-your-application/rendering/server-components).

## `headers()`

This API extends the [Web Headers API](https://developer.mozilla.org/docs/Web/API/Headers). It is **read-only**, meaning you cannot `set` / `delete` the outgoing request headers.

```tsx filename="app/page.tsx" switcher
import { headers } from 'next/headers'

export default function Page() {
  const headersList = headers()
  const referer = headersList.get('referer')

  return <div>Referer: {referer}</div>
}
```

```jsx filename="app/page.js" switcher
import { headers } from 'next/headers'

export default function Page() {
  const headersList = headers()
  const referer = headersList.get('referer')

  return <div>Referer: {referer}</div>
}
```

> **Good to know**:

>
> - `headers()` is a **[Dynamic Function](/docs/app/building-your-application/rendering/server-components#server-rendering-strategies#dynamic-functions)** whose returned values cannot be known ahead of time. Using it in a layout or page will opt a route into **[dynamic rendering](/docs/app/building-your-application/rendering/server-components#dynamic-rendering)** at request time.

### API Reference

```tsx
const headersList = headers()
```

#### Parameters

`headers` does not take any parameters.

#### Returns

`headers` returns a **read-only** [Web Headers](https://developer.mozilla.org/docs/Web/API/Headers) object.

- [`Headers.entries()`](https://developer.mozilla.org/docs/Web/API/Headers/entries): Returns an [`iterator`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Iteration_protocols) allowing to go through all key/value pairs contained in this object.
- [`Headers.forEach()`](https://developer.mozilla.org/docs/Web/API/Headers/forEach): Executes a provided function once for each key/value pair in this `Headers` object.
- [`Headers.get()`](https://developer.mozilla.org/docs/Web/API/Headers/get): Returns a [`String`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String) sequence of all the values of a header within a `Headers` object with a given name.
- [`Headers.has()`](https://developer.mozilla.org/docs/Web/API/Headers/has): Returns a boolean stating whether a `Headers` object contains a certain header.
- [`Headers.keys()`](https://developer.mozilla.org/docs/Web/API/Headers/keys): Returns an [`iterator`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Iteration_protocols) allowing you to go through all keys of the key/value pairs contained in this object.
- [`Headers.values()`](https://developer.mozilla.org/docs/Web/API/Headers/values): Returns an [`iterator`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Iteration_protocols) 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](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating).

```jsx filename="app/page.js"
import { headers } from 'next/headers'

async function getUser() {
  const headersInstance = headers()
  const authorization = headersInstance.get('authorization')
  // Forward the authorization header
  const res = await fetch('...', {
    headers: { authorization },
  })
  return res.json()
}

export default async function UserPage() {
  const user = await getUser()
  return <h1>{user.name}</h1>
}
```

## Version History

Version	Changes
`v13.0.0`	`headers` introduced.

---
title: ImageResponse
description: API Reference for the ImageResponse constructor.

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`:

```jsx
import { ImageResponse } from 'next/og'

new ImageResponse(
  element: ReactElement,
  options: {
    width?: number = 1200
    height?: number = 630
    emoji?: 'twemoji' | 'blobmoji' | 'noto' | 'openmoji' = 'twemoji',
    fonts?: {
      name: string,
      data: ArrayBuffer,
      weight: number,
      style: 'normal' | 'italic'
    }[]
    debug?: boolean = false

    // Options that will be passed to the HTTP response
    status?: number = 200
    statusText?: string
    headers?: Record<string, string>
  },
)
```

## Supported CSS Properties

Please refer to [Satori’s documentation](https://github.com/vercel/satori#css) for a list of supported HTML and CSS features.

## Version History

Version	Changes
`v14.0.0`	`ImageResponse` moved from `next/server` to `next/og`
`v13.3.0`	`ImageResponse` can be imported from `next/server`.
`v13.0.0`	`ImageResponse` introduced via `@vercel/og` package.

---
title: Functions
description: API Reference for Next.js Functions and Hooks.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

---
title: NextRequest
description: API Reference for NextRequest.

NextRequest extends the [Web Request API](https://developer.mozilla.org/docs/Web/API/Request) with additional convenience methods.

## `cookies`

Read or mutate the [`Set-Cookie`](https://developer.mozilla.org/docs/Web/HTTP/Headers/Set-Cookie) header of the request.

### `set(name, value)`

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

```ts
// Given incoming request /home
// Set a cookie to hide the banner
// request will have a `Set-Cookie:show-banner=false;path=/home` header
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.

```ts
// Given incoming request /home
// { name: 'show-banner', value: 'false', Path: '/home' }
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.

```ts
// Given incoming request /home
// [
//   { name: 'experiments', value: 'new-pricing-page', Path: '/home' },
//   { name: 'experiments', value: 'winter-launch', Path: '/home' },
// ]
request.cookies.getAll('experiments')
// Alternatively, get all cookies for the request
request.cookies.getAll()
```

### `delete(name)`

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

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

### `has(name)`

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

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

### `clear()`

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

```ts
request.cookies.clear()
```

## `nextUrl`

Extends the native [`URL`](https://developer.mozilla.org/docs/Web/API/URL) API with additional convenience methods, including Next.js specific properties.

```ts
// Given a request to /home, pathname is /home
request.nextUrl.pathname
// Given a request to /home?name=lee, searchParams is { 'name': 'lee' }
request.nextUrl.searchParams
```

## Version History

Version	Changes
`v13.0.0`	`useSearchParams` introduced.

---
title: NextResponse

description: API Reference for NextResponse.

NextResponse extends the [Web Response API](https://developer.mozilla.org/docs/Web/API/Response) with additional convenience methods.

## `cookies`

Read or mutate the [`Set-Cookie`](https://developer.mozilla.org/docs/Web/HTTP/Headers/Set-Cookie) header of the response.

### `set(name, value)`

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

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

```ts
// Given incoming request /home
let response = NextResponse.next()
// { name: 'show-banner', value: 'false', Path: '/home' }
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.

```ts
// Given incoming request /home
let response = NextResponse.next()
// [
//   { name: 'experiments', value: 'new-pricing-page', Path: '/home' },
//   { name: 'experiments', value: 'winter-launch', Path: '/home' },
```

//]
response.cookies.getAll('experiments')
// Alternatively, get all cookies for the response
response.cookies.getAll()

### `delete(name)`

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

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

## `json()`

Produce a response with the given JSON body.

```ts filename="app/api/route.ts" switcher
import { NextResponse } from 'next/server'

export async function GET(request: Request) {
  return NextResponse.json({ error: 'Internal Server Error' }, { status: 500 })
}
```

```js filename="app/api/route.js" switcher
import { NextResponse } from 'next/server'

export async function GET(request) {
  return NextResponse.json({ error: 'Internal Server Error' }, { status: 500 })
}
```

## `redirect()`

Produce a response that redirects to a [URL](https://developer.mozilla.org/docs/Web/API/URL).

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

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

The [URL](https://developer.mozilla.org/docs/Web/API/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.

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

// Given an incoming request...
const loginUrl = new URL('/login', request.url)
// Add ?from=/incoming-url to the /login URL
loginUrl.searchParams.set('from', request.nextUrl.pathname)
// And redirect to the new URL
return NextResponse.redirect(loginUrl)
```

## `rewrite()`

Produce a response that rewrites (proxies) the given [URL](https://developer.mozilla.org/docs/Web/API/URL) while preserving the original URL.

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

// Incoming request: /about, browser shows /about
// Rewritten request: /proxy, browser shows /about
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.

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

return NextResponse.next()
```

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

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

// Given an incoming request...
```

```
const newHeaders = new Headers(request.headers)
// Add a new header
newHeaders.set('x-version', '123')
// And produce a response with the new headers
return NextResponse.next({
  request: {
    // New request headers
    headers: newHeaders,
  },
})
```

---
title: notFound
description: API Reference for the notFound function.

The `notFound` function allows you to render the [`not-found file`](/docs/app/api-reference/file-conventions/not-found) 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](/docs/app/api-reference/file-conventions/not-found) allows you to gracefully handle such errors by rendering a Not Found UI within the segment.

```jsx filename="app/user/[id]/page.js"
import { notFound } from 'next/navigation'

async function fetchUser(id) {
  const res = await fetch('https://...')
  if (!res.ok) return undefined
  return res.json()
}

export default async function Profile({ params }) {
  const user = await fetchUser(params.id)

  if (!user) {
    notFound()
  }

  // ...
}
```

> **Good to know**: `notFound()` does not require you to use `return notFound()` due to using the TypeScript [`never`](https://www.typescriptlang.org/docs/handbook/2/functions.html#never) type.

## Version History

Version	Changes
`v13.0.0`	`notFound` introduced.

---
title: permanentRedirect
description: API Reference for the permanentRedirect function.

The `permanentRedirect` function allows you to redirect the user to another URL. `permanentRedirect` can be used in Server Components, Client Components, [Route Handlers](/docs/app/building-your-application/routing/route-handlers), and [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations).

When used in a streaming context, this will insert a meta tag to emit the redirect on the client side. When used in a server action, it will serve a 303 HTTP redirect response to the caller. Otherwise, it will serve a 308 (Permanent) HTTP redirect response to the caller.

If a resource doesn't exist, you can use the [`notFound` function](/docs/app/api-reference/functions/not-found) instead.

> **Good to know**: If you prefer to return a 307 (Temporary) HTTP redirect instead of 308 (Permanent), you can use the [`redirect` function](/docs/app/api-reference/functions/redirect) instead.

## Parameters

The `permanentRedirect` function accepts two arguments:

```js
permanentRedirect(path, type)
```

| Parameter | Type | Description
|
| --------- | --- |
--- |

| `path` | `string` | The URL to redirect to. Can be a relative or absolute path. |
| `type` | `'replace'` (default) or `'push'` (default in Server Actions) | The type of redirect to perform. |

By default, `permanentRedirect` will use `push` (adding a new entry to the browser history stack) in [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations) and `replace` (replacing the current URL in the browser history stack) everywhere else. You can override this behavior by specifying the `type` parameter.

The `type` parameter has no effect when used in Server Components.

## Returns

`permanentRedirect` does not return any value.

## Example

Invoking the `permanentRedirect()` function throws a `NEXT_REDIRECT` error and terminates rendering of the route segment in which it was thrown.

```jsx filename="app/team/[id]/page.js"
import { permanentRedirect } from 'next/navigation'

async function fetchTeam(id) {
  const res = await fetch('https://...')
  if (!res.ok) return undefined
  return res.json()
}

export default async function Profile({ params }) {
  const team = await fetchTeam(params.id)
  if (!team) {
    permanentRedirect('/login')
  }

  // ...
}
```

> **Good to know**: `permanentRedirect` does not require you to use `return permanentRedirect()` as it uses the TypeScript [`never`](https://www.typescriptlang.org/docs/handbook/2/functions.html#never) type.

---
title: redirect

description: API Reference for the redirect function.

The `redirect` function allows you to redirect the user to another URL. `redirect` can be used in Server Components, Client Components, [Route Handlers](/docs/app/building-your-application/routing/route-handlers), and [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations).

When used in a [streaming context](/docs/app/building-your-application/routing/loading-ui-and-streaming#what-is-streaming), this will insert a meta tag to emit the redirect on the client side. When used in a server action, it will serve a 303 HTTP redirect response to the caller. Otherwise, it will serve a 307 HTTP redirect response to the caller.

If a resource doesn't exist, you can use the [`notFound` function](/docs/app/api-reference/functions/not-found) instead.

> **Good to know**: If you prefer to return a 308 (Permanent) HTTP redirect instead of 307 (Temporary), you can use the [`permanentRedirect` function](/docs/app/api-reference/functions/permanentRedirect) instead.

## Parameters

The `redirect` function accepts two arguments:

```js
redirect(path, type)
```

Parameter	Type	Description
`path`	`string`	The URL to redirect to. Can be a relative or absolute path.
`type`	`'replace'` (default) or `'push'` (default in Server Actions)	The type of redirect to perform.

By default, `redirect` will use `push` (adding a new entry to the browser history stack) in [Server Actions](/docs/app/building-your-application/data-fetching/forms-and-mutations) and `replace` (replacing the current URL in the browser history stack) everywhere else. You can override this behavior by specifying the `type` parameter.

The `type` parameter has no effect when used in Server Components.

## Returns

`redirect` does not return any value.

## Example

Invoking the `redirect()` function throws a `NEXT_REDIRECT` error and terminates rendering of the route segment in which it was thrown.

> **Good to know**: If you need to programmatically redirect the user after a certain event in a Client Component, you can use the [`useRouter`](/docs/app/api-reference/functions/use-router) hook.

```jsx filename="app/team/[id]/page.js"
import { redirect } from 'next/navigation'

async function fetchTeam(id) {
  const res = await fetch('https://...')
  if (!res.ok) return undefined
  return res.json()
}

export default async function Profile({ params }) {
  const team = await fetchTeam(params.id)
  if (!team) {
    redirect('/login')
  }

  // ...
}
```

> **Good to know**: `redirect` does not require you to use `return redirect()` as it uses the TypeScript [`never`](https://www.typescriptlang.org/docs/handbook/2/functions.html#never) type.

Version	Changes
`v13.0.0`	`redirect` introduced.
---
title: revalidatePath
description: API Reference for the revalidatePath function.

`revalidatePath` allows you to purge [cached data](/docs/app/building-your-application/caching) on-demand for a specific path.

> **Good to know**:
>
> - `revalidatePath` is available in both [Node.js and Edge runtimes](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes).
> - `revalidatePath` only invalidates the cache when the included 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

```tsx
revalidatePath(path: string, type?: 'page' | 'layout'): void;
```

- `path`: Either a string representing the filesystem path associated with the data you want to revalidate (for example, `/product/[slug]/page`), or the literal route segment (for example, `/product/123`). Must be less than 1024 characters.
- `type`: (optional) `'page'` or `'layout'` string to change the type of path to revalidate.

## Returns

`revalidatePath` does not return any value.

## Examples

### Revalidating A Specific URL

```ts
import { revalidatePath } from 'next/cache'
revalidatePath('/blog/post-1')
```

This will revalidate one specific URL on the next page visit.

### Revalidating A Page Path

```ts
import { revalidatePath } from 'next/cache'
revalidatePath('/blog/[slug]', 'page')
// or with route groups
revalidatePath('/(main)/post/[slug]', 'page')
```

This will revalidate any URL that matches the provided `page` file on the next page visit. This will _not_ invalidate pages beneath the specific page. For example, `/blog/[slug]` won't invalidate `/blog/[slug]/[author]`.

### Revalidating A Layout Path

```ts
import { revalidatePath } from 'next/cache'
revalidatePath('/blog/[slug]', 'layout')
// or with route groups
revalidatePath('/(main)/post/[slug]', 'layout')
```

This will revalidate any URL that matches the provided `layout` file on the next page visit. This will cause pages beneath with the same layout to revalidate on the next visit. For example, in the above case, `/blog/[slug]/[another]` would also revalidate on the next visit.

### Revalidating All Data

```ts
import { revalidatePath } from 'next/cache'

revalidatePath('/', 'layout')
```

This will purge the Client-side Router Cache, and revalidate the Data Cache on the next page visit.

### Server Action

```ts filename="app/actions.ts" switcher
'use server'

import { revalidatePath } from 'next/cache'

export default async function submit() {
  await submitForm()
  revalidatePath('/')
}
```

### Route Handler

```ts filename="app/api/revalidate/route.ts" switcher
import { revalidatePath } from 'next/cache'
import { NextRequest } from 'next/server'

export async function GET(request: NextRequest) {
  const path = request.nextUrl.searchParams.get('path')

  if (path) {
    revalidatePath(path)
    return Response.json({ revalidated: true, now: Date.now() })
  }

  return Response.json({
    revalidated: false,
    now: Date.now(),
    message: 'Missing path to revalidate',
  })
}
```

```js filename="app/api/revalidate/route.js" switcher
import { revalidatePath } from 'next/cache'

export async function GET(request) {
  const path = request.nextUrl.searchParams.get('path')

  if (path) {
    revalidatePath(path)
    return Response.json({ revalidated: true, now: Date.now() })
  }

  return Response.json({
    revalidated: false,
    now: Date.now(),
    message: 'Missing path to revalidate',
  })
}
```

---
title: revalidateTag
description: API Reference for the revalidateTag function.

`revalidateTag` allows you to purge [cached data](/docs/app/building-your-application/caching) on-demand for a specific cache tag.

> **Good to know**:
>

> - `revalidateTag` is available in both [Node.js and Edge runtimes](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes).
> - `revalidateTag` only invalidates the cache when the path is next visited. This means calling `revalidateTag` with a dynamic route segment will not immediately trigger many revalidations at once. The invalidation only happens when the path is next visited.

## Parameters

```tsx
revalidateTag(tag: string): void;
```

- `tag`: A string representing the cache tag associated with the data you want to revalidate. Must be less than or equal to 256 characters.

You can add tags to `fetch` as follows:

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

## Returns

`revalidateTag` does not return any value.

## Examples

### Server Action

```ts filename="app/actions.ts" switcher
'use server'

import { revalidateTag } from 'next/cache'

export default async function submit() {
  await addPost()
  revalidateTag('posts')
}
```

```js filename="app/actions.js" switcher
'use server'

import { revalidateTag } from 'next/cache'

export default async function submit() {
```

  await addPost()
  revalidateTag('posts')
}

### Route Handler

```ts filename="app/api/revalidate/route.ts" switcher
import { NextRequest } from 'next/server'
import { revalidateTag } from 'next/cache'

export async function GET(request: NextRequest) {
  const tag = request.nextUrl.searchParams.get('tag')
  revalidateTag(tag)
  return Response.json({ revalidated: true, now: Date.now() })
}
```

```js filename="app/api/revalidate/route.js" switcher
import { revalidateTag } from 'next/cache'

export async function GET(request) {
  const tag = request.nextUrl.searchParams.get('tag')
  revalidateTag(tag)
  return Response.json({ revalidated: true, now: Date.now() })
}
```

---
title: Server Actions
nav_title: Server Actions
description: API Reference for Next.js Server Actions.
related:
  title: Next Steps
  description: For more information on what to do next, we recommend the following sections
  links:
    - app/building-your-application/data-fetching/forms-and-mutations
---

{/* TODO: This page will need to link back to React docs mentioned at the bottom */}

Next.js integrates with React Actions to provide a built-in solution for [server mutations](/docs/app/building-your-application/data-fetching/forms-and-mutations).

## Convention

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"`](https://react.dev/reference/react/use-server) directive at the top of the function body. `"use server"` ensures this function is only ever executed on the server.

This function should have [serializable arguments](https://developer.mozilla.org/docs/Glossary/Serialization) and a [serializable return value](https://developer.mozilla.org/docs/Glossary/Serialization).

```jsx filename="app/page.js" highlight={2}
export default function ServerComponent() {
  async function myAction() {
    'use server'
    // ...
  }
}
```

### With Client Components

#### Import

Create your Server 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:

```js filename="app/actions.js" highlight={1}
'use server'

export async function myAction() {
  // ...
}
```

```jsx filename="app/client-component.jsx" highlight={1}
'use client'
```

import { myAction } from './actions'

export default function ClientComponent() {
  return (
    <form action={myAction}>
      <button type="submit">Add to Cart</button>
    </form>
)
}

> **Good to know**: 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.

#### Props

In some cases, you might want to pass down a Server Action to a Client Component as a prop.

```jsx
<ClientComponent myAction={updateItem} />
```

```jsx filename="app/client-component.jsx"
'use client'

export default function ClientComponent({ myAction }) {
  return (
    <form action={myAction}>
      <input type="text" name="name" />
      <button type="submit">Update Item</button>
    </form>
  )
}
```

### Binding Arguments

You can bind arguments to a Server Action using the `bind` method. This allows you to create a new Server Action with some arguments already bound. This is beneficial when you want to pass extra arguments to a Server Action.

```jsx filename="app/client-component.jsx" highlight={6}
'use client'
```

```
import { updateUser } from './actions'

export function UserProfile({ userId }) {
  const updateUserWithId = updateUser.bind(null, userId)

  return (
    <form action={updateUserWithId}>
      <input type="text" name="name" />
      <button type="submit">Update User Name</button>
    </form>
  )
}
```

And then, the `updateUser` Server Action will always receive the `userId` argument, in addition to the form data:

```js filename="app/actions.js"
'use server'

export async function updateUser(userId, formData) {
  // ...
}
```

> **Good to know**: `.bind` of a Server Action works in both Server and Client Components. It also supports [Progressive Enhancement](#progressive-enhancement).

## 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](#progressive-enhancement)**.

## 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.

React Actions (both server and client) 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. React prioritizes the action, so it still happens quickly.

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.

## Size Limitation

By default, the maximum size of the request body sent to a Server Action is 1MB, to prevent the consumption of excessive server resources in parsing large amounts of data.

However, you can configure this limit using the `serverActions.bodySizeLimit` option. It can take the number of bytes or any string format supported by bytes, for example `1000`, `'500kb'` or `'3mb'`.

```js filename="next.config.js"
module.exports = {
  experimental: {
    serverActions: {
      bodySizeLimit: '2mb',
    },
  },
}
```

## Additional Resources

The following React API pages are currently being documented:

- [`"use server"`](https://react.dev/reference/react/use-server)
- [`useFormStatus`](https://react.dev/reference/react-dom/hooks/useFormStatus)
- [`useFormState`](https://react.dev/reference/react-dom/hooks/useFormState)
- [`useOptimistic`](https://react.dev/reference/react/useOptimistic)
- [`<form>`](https://react.dev/reference/react-dom/components/form)

---
title: unstable_cache
description: API Reference for the unstable_cache function.

`unstable_cache` allows you to cache the results of expensive operations, like database queries, and reuse them across multiple requests.

```jsx
import { getUser } from './data';
import { unstable_cache } from 'next/cache';

const getCachedUser = unstable_cache(
  async (id) => getUser(id),
  ['my-app-user']
);

export default async function Component({ userID }) {
  const user = await getCachedUser(userID);
  ...
}
```

> **Warning**: This API is unstable and may change in the future. We will provide migration documentation and codemods, if needed, as this API stabilizes.

## Parameters

```jsx
const data = unstable_cache(fetchData, keyParts, options)()
```

- `fetchData`: This is an asynchronous function that fetches the data you want to cache. It must be a function that returns a `Promise`.
- `keyParts`: This is an array that identifies the cached key. It must contain globally unique values that together identify the key of the data being cached. The cache key also includes the arguments passed to the function.
- `options`: This is an object that controls how the cache behaves. It can contain the following properties:
  - `tags`: An array of tags that can be used to control cache invalidation.
  - `revalidate`: The number of seconds after which the cache should be revalidated.

## Returns

`unstable_cache` returns a function that when invoked, returns a Promise that resolves to the cached data. If the data is not in the cache, the provided function will be invoked, and its result will be cached and returned.

title: unstable_noStore
description: API Reference for the unstable_noStore function.

`unstable_noStore` can be used to declaratively opt out of static rendering and indicate a particular component should not be cached.

```jsx
import { unstable_noStore as noStore } from 'next/cache';

export default async function Component() {
  noStore();
  const result = await db.query(...);
  ...
}
```

> **Good to know**:
>
> - `unstable_noStore` is equivalent to `cache: 'no-store'` on a `fetch`
> - `unstable_noStore` is preferred over `export const dynamic = 'force-dynamic'` as it is more granular and can be used on a per-component basis

- Using `unstable_noStore` inside [`unstable_cache`](/docs/app/api-reference/functions/unstable_cache) will not opt out of static generation. Instead, it will defer to the cache configuration to determine whether to cache the result or not.

## Usage

If you prefer not to pass additional options to `fetch`, like `cache: 'no-store'` or `next: { revalidate: 0 }`, you can use `noStore()` as a replacement for all of these use cases.

```jsx
import { unstable_noStore as noStore } from 'next/cache';

export default async function Component() {
  noStore();
  const result = await db.query(...);
  ...
```

}

## Version History

Version	Changes
`v14.0.0`	`unstable_noStore` introduced.

---
title: useParams
description: API Reference for the useParams hook.

`useParams` is a **Client Component** hook that lets you read a route's [dynamic params](/docs/app/building-your-application/routing/dynamic-routes) filled in by the current URL.

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { useParams } from 'next/navigation'

export default function ExampleClientComponent() {
  const params = useParams()

  // Route -> /shop/[tag]/[item]
  // URL -> /shop/shoes/nike-air-max-97
  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
  console.log(params)

  return <></>
}
```

```jsx filename="app/example-client-component.js" switcher
'use client'

import { useParams } from 'next/navigation'

export default function ExampleClientComponent() {
  const params = useParams()

  // Route -> /shop/[tag]/[item]
  // URL -> /shop/shoes/nike-air-max-97
  // `params` -> { tag: 'shoes', item: 'nike-air-max-97' }
  console.log(params)
  return <></>
}
```

## Parameters

```tsx
const params = useParams()
```

`useParams` does not take any parameters.

## Returns

`useParams` returns an object containing the current route's filled in [dynamic parameters](/docs/app/building-your-application/routing/dynamic-routes).

- 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](/docs/app/building-your-application/routing/dynamic-routes).
- If the route contains no dynamic parameters, `useParams` returns an empty object.
- If used in `pages`, `useParams` will return `null`.

For example:

Route	URL	`useParams()`
`app/shop/page.js`	`/shop`	`null`
`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'] }`

## Version History

Version	Changes
`v13.3.0`	`useParams` introduced.

---
title: usePathname
description: API Reference for the usePathname hook.

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

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { usePathname } from 'next/navigation'

export default function ExampleClientComponent() {
  const pathname = usePathname()
  return <p>Current pathname: {pathname}</p>
}
```

```jsx filename="app/example-client-component.js" switcher
'use client'

import { usePathname } from 'next/navigation'

export default function ExampleClientComponent() {
  const pathname = usePathname()
  return <p>Current pathname: {pathname}</p>
}
```

`usePathname` intentionally requires using a [Client Component](/docs/app/building-your-application/rendering/client-components). It's important to note Client Components are not a de-optimization. They are an integral part of the [Server Components](/docs/app/building-your-application/rendering/server-components) architecture.

For example, a Client Component with `usePathname` will be rendered into HTML on the initial page load. When navigating to a new route, this component does not need to be re-fetched. Instead, the component is downloaded once (in the client JavaScript bundle), and re-renders based on the current state.

> **Good to know**:
>
> - Reading the current URL from a [Server Component](/docs/app/building-your-application/rendering/server-components) is not supported. This design is intentional to support layout state being preserved across page navigations.
> - Compatibility mode:
>   - `usePathname` can return `null` when a [fallback route](/docs/pages/api-reference/functions/get-static-paths#fallback-true) is being rendered or when a `pages` directory page has been [automatically statically optimized](/docs/pages/building-your-application/rendering/automatic-static-optimization) 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

```tsx
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

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { usePathname, useSearchParams } from 'next/navigation'

function ExampleClientComponent() {
  const pathname = usePathname()
  const searchParams = useSearchParams()
  useEffect(() => {
    // Do something here...
  }, [pathname, searchParams])
}
```

```jsx filename="app/example-client-component.js" switcher
'use client'

import { usePathname, useSearchParams } from 'next/navigation'

function ExampleClientComponent() {
```

  const pathname = usePathname()
  const searchParams = useSearchParams()
  useEffect(() => {
    // Do something here...
  }, [pathname, searchParams])
}

Version	Changes
`v13.0.0`	`usePathname` introduced.

---
title: useReportWebVitals
description: API Reference for the useReportWebVitals function.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

The `useReportWebVitals` hook allows you to report [Core Web Vitals](https://web.dev/vitals/), and can be used in combination with your analytics service.

<PagesOnly>

```jsx filename="pages/_app.js"
import { useReportWebVitals } from 'next/web-vitals'

function MyApp({ Component, pageProps }) {
  useReportWebVitals((metric) => {
    console.log(metric)
  })

  return <Component {...pageProps} />
}
```

</PagesOnly>

<AppOnly>

```jsx filename="app/_components/web-vitals.js"
'use client'

import { useReportWebVitals } from 'next/web-vitals'
```

```
export function WebVitals() {
  useReportWebVitals((metric) => {
    console.log(metric)
  })
}
```

```jsx filename="app/layout.js"
import { WebVitals } from './_components/web-vitals'

export default function Layout({ children }) {
  return (
    <html>
      <body>
        <WebVitals />
        {children}
      </body>
    </html>
  )
}
```

> 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.

</AppOnly>

## 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](#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](https://developer.mozilla.org/docs/Web/API/PerformanceEntry) associated with the metric. These entries provide detailed information about the performance events related to the metric.
- `navigationType`: Indicates the [type of navigation](https://

developer.mozilla.org/docs/Web/API/PerformanceNavigationTiming/type) 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](https://developer.mozilla.org/docs/Web/API/Performance_API)s.

## Web Vitals

[Web Vitals](https://web.dev/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](https://developer.mozilla.org/docs/Glossary/Time_to_first_byte) (TTFB)
- [First Contentful Paint](https://developer.mozilla.org/docs/Glossary/First_contentful_paint) (FCP)
- [Largest Contentful Paint](https://web.dev/lcp/) (LCP)
- [First Input Delay](https://web.dev/fid/) (FID)
- [Cumulative Layout Shift](https://web.dev/cls/) (CLS)
- [Interaction to Next Paint](https://web.dev/inp/) (INP)

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

<PagesOnly>

```jsx filename="pages/_app.js"
import { useReportWebVitals } from 'next/web-vitals'

function MyApp({ Component, pageProps }) {
  useReportWebVitals((metric) => {
    switch (metric.name) {
      case 'FCP': {
        // handle FCP results
      }
      case 'LCP': {
        // handle LCP results
      }
      // ...
    }
```

~~~
  })

  return <Component {...pageProps} />
}
```

</PagesOnly>

<AppOnly>

```tsx filename="app/components/web-vitals.tsx" switcher
'use client'

import { useReportWebVitals } from 'next/web-vitals'

export function WebVitals() {
  useReportWebVitals((metric) => {
    switch (metric.name) {
      case 'FCP': {
        // handle FCP results
      }
      case 'LCP': {
        // handle LCP results
      }
      // ...
    }
  })
}
```

```jsx filename="app/components/web-vitals.js" switcher
'use client'

import { useReportWebVitals } from 'next/web-vitals'

export function WebVitals() {
  useReportWebVitals((metric) => {
    switch (metric.name) {
      case 'FCP': {
        // handle FCP results
      }
      case 'LCP': {
        // handle LCP results
      }
      // ...
    }
  })
~~~

```
}
```

</AppOnly>

<PagesOnly>

## Custom Metrics

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

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

You can handle all the results of these metrics separately:

```js
export function reportWebVitals(metric) {
  switch (metric.name) {
    case 'Next.js-hydration':
      // handle hydration results
      break
    case 'Next.js-route-change-to-render':
      // handle route-change to render results
      break
    case 'Next.js-render':
      // handle render results
      break
    default:
      break
  }
}
```

These metrics work in all browsers that support the [User Timing API](https://caniuse.com/#feat=user-timing).

</PagesOnly>

## Usage on Vercel

[Vercel Speed Insights](https://vercel.com/docs/concepts/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:

```js
useReportWebVitals((metric) => {
  const body = JSON.stringify(metric)
  const url = 'https://example.com/analytics'

  // Use `navigator.sendBeacon()` if available, falling back to `fetch()`.
  if (navigator.sendBeacon) {
    navigator.sendBeacon(url, body)
  } else {
    fetch(url, { body, method: 'POST', keepalive: true })
  }
})
```

> **Good to know**: If you use [Google Analytics](https://analytics.google.com/analytics/web/), using the
> `id` value can allow you to construct metric distributions manually (to calculate percentiles,
> etc.)

> ```js
> useReportWebVitals(metric => {
>   // Use `window.gtag` if you initialized Google Analytics as this example:
>   // https://github.com/vercel/next.js/blob/canary/examples/with-google-analytics/pages/_app.js
>   window.gtag('event', metric.name, {
>     value: Math.round(metric.name === 'CLS' ? metric.value * 1000 : metric.value), // values must be integers
>     event_label: metric.id, // id unique to current page load
>     non_interaction: true, // avoids affecting bounce rate.
>   });
> }
> ```
>
> Read more about [sending results to Google Analytics](https://github.com/GoogleChrome/web-vitals#send-the-results-to-google-analytics).

---
title: useRouter
description: API reference for the useRouter hook.

The `useRouter` hook allows you to programmatically change routes inside [Client Components](/docs/app/building-your-application/rendering/client-components).

> **Recommendation:** Use the [`<Link>` component](/docs/app/building-your-application/routing/linking-and-navigating#link-component) for navigation unless you have a specific requirement for using `useRouter`.

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```

```jsx filename="app/example-client-component.js" switcher
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```

## `useRouter()`

- `router.push(href: string, { scroll: boolean })`: Perform a client-side navigation to the provided route. Adds a new entry into the [browser’s history](https://developer.mozilla.org/docs/Web/API/History_API) stack.
- `router.replace(href: string, { scroll: boolean })`: Perform a client-side navigation to the provided route without adding a new entry into the [browser’s history stack](https://developer.mozilla.org/docs/Web/API/History_API).
- `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](/docs/app/building-your-application/routing/linking-and-navigating#1-prefetching) the provided route for faster client-side transitions.
- `router.back()`: Navigate back to the previous route in the browser’s history stack.
- `router.forward()`: Navigate forwards to the next page in the browser’s history stack.

> **Good to know**:
>
> - The `<Link>` component 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 `next/router`

- The `useRouter` hook should be imported from `next/navigation` and not `next/router` when using the App Router
- The `pathname` string has been removed and is replaced by [`usePathname()`](/docs/app/api-reference/functions/use-pathname)
- The `query` object has been removed and is replaced by [`useSearchParams()`](/docs/app/api-reference/functions/use-search-params)
- `router.events` has been replaced. [See below.](#router-events)

[View the full migration guide](/docs/app/building-your-application/upgrading/app-router-migration).

## Examples

### Router events

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

```jsx filename="app/components/navigation-events.js"
'use client'

import { useEffect } from 'react'
import { usePathname, useSearchParams } from 'next/navigation'

export function NavigationEvents() {
  const pathname = usePathname()
  const searchParams = useSearchParams()

  useEffect(() => {
    const url = `${pathname}?${searchParams}`
    console.log(url)
    // You can now use the current URL
    // ...
  }, [pathname, searchParams])

  return null
}
```

Which can be imported into a layout.

```jsx filename="app/layout.js" highlight={2,10-12}
import { Suspense } from 'react'
import { NavigationEvents } from './components/navigation-events'

export default function Layout({ children }) {
  return (
    <html lang="en">
      <body>
        {children}

        <Suspense fallback={null}>
          <NavigationEvents />
        </Suspense>
      </body>
    </html>
  )
}
```

> **Good to know**: `<NavigationEvents>` is wrapped in a [`Suspense` boundary](/docs/app/building-your-application/routing/loading-ui-and-streaming#example) because[`useSearchParams()`](/docs/app/api-reference/functions/use-search-params) causes client-side rendering up to the closest

`Suspense` boundary during [static rendering](/docs/app/building-your-application/rendering/server-components#static-rendering-default). [Learn more](/docs/app/api-reference/functions/use-search-params#behavior).

### Disabling scroll restoration

By default, Next.js will scroll to the top of the page when navigating to a new route. You can disable this behavior by passing `scroll: false` to `router.push()` or `router.replace()`.

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button
      type="button"
      onClick={() => router.push('/dashboard', { scroll: false })}
    >
      Dashboard
    </button>
  )
}
```

```jsx filename="app/example-client-component.jsx" switcher
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button
      type="button"
      onClick={() => router.push('/dashboard', { scroll: false })}
    >
      Dashboard
    </button>
  )
}
```

## Version History

Version	Changes
`v13.0.0`	`useRouter` from `next/navigation` introduced.

---
title: useSearchParams
description: API Reference for the useSearchParams hook.

`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`](https://developer.mozilla.org/docs/Web/API/URLSearchParams) interface.

```tsx filename="app/dashboard/search-bar.tsx" switcher
'use client'

import { useSearchParams } from 'next/navigation'

export default function SearchBar() {
  const searchParams = useSearchParams()

  const search = searchParams.get('search')

  // URL -> `/dashboard?search=my-project`
  // `search` -> 'my-project'
  return <>Search: {search}</>
}
```

```jsx filename="app/dashboard/search-bar.js" switcher
'use client'

import { useSearchParams } from 'next/navigation'

export default function SearchBar() {
  const searchParams = useSearchParams()

  const search = searchParams.get('search')

  // URL -> `/dashboard?search=my-project`
  // `search` -> 'my-project'
```

  return <>Search: {search}</>
}

## Parameters

```tsx
const searchParams = useSearchParams()
```

`useSearchParams` does not take any parameters.

## Returns

`useSearchParams` returns a **read-only** version of the [`URLSearchParams`](https://developer.mozilla.org/docs/Web/API/URLSearchParams) interface, which includes utility methods for reading the URL's query string:

- [`URLSearchParams.get()`](https://developer.mozilla.org/docs/Web/API/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()`](https://developer.mozilla.org/docs/Web/API/URLSearchParams/getAll) to get all values_

- [`URLSearchParams.has()`](https://developer.mozilla.org/docs/Web/API/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`](https://developer.mozilla.org/docs/Web/API/URLSearchParams), including the [`getAll()`](https://developer.mozilla.org/docs/Web/API/URLSearchParams/getAll), [`keys()`](https://developer.mozilla.org/docs/Web/API/URLSearchParams/keys), [`values()`](https://developer.mozilla.org/docs/Web/API/URLSearchParams/values), [`entries()`](https://developer.mozilla.org/docs/Web/API/URLSearchParams/entries), [`forEach()`](https://developer.mozilla.org/docs/Web/API/URLSearchParams/forEach), and [`toString()`](https://developer.mozilla.org/docs/Web/API/URLSearchParams/toString).

> **Good to know**:
>
> - `useSearchParams` is a [Client Component](/docs/app/building-your-application/rendering/client-components) hook and is **not supported** in [Server Components](/docs/app/building-your-application/rendering/server-components) to prevent stale values during [partial rendering](/docs/app/building-your-application/routing/linking-and-navigating#3-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](/docs/app/building-your-application/rendering/server-components#static-rendering-default), calling `useSearchParams()` will cause the tree up to the closest [`Suspense` boundary](/docs/app/building-your-application/routing/loading-ui-and-streaming#example) 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:

```tsx filename="app/dashboard/search-bar.tsx" switcher
'use client'

import { useSearchParams } from 'next/navigation'

export default function SearchBar() {
```

~~~
  const searchParams = useSearchParams()

  const search = searchParams.get('search')

  // This will not be logged on the server when using static rendering
  console.log(search)

  return <>Search: {search}</>
}
```

```jsx filename="app/dashboard/search-bar.js" switcher
'use client'

import { useSearchParams } from 'next/navigation'

export default function SearchBar() {
  const searchParams = useSearchParams()

  const search = searchParams.get('search')

  // This will not be logged on the server when using static rendering
  console.log(search)

  return <>Search: {search}</>
}
```

```tsx filename="app/dashboard/page.tsx" switcher
import { Suspense } from 'react'
import SearchBar from './search-bar'

// This component passed as a fallback to the Suspense boundary
// will be rendered in place of the search bar in the initial HTML.
// When the value is available during React hydration the fallback
// will be replaced with the `<SearchBar>` component.
function SearchBarFallback() {
  return <>placeholder</>
}

export default function Page() {
  return (
    <>
      <nav>
        <Suspense fallback={<SearchBarFallback />}>
          <SearchBar />
        </Suspense>
~~~

      </nav>
      <h1>Dashboard</h1>
    </>
)
}

```jsx filename="app/dashboard/page.js" switcher
import { Suspense } from 'react'
import SearchBar from './search-bar'

// This component passed as a fallback to the Suspense boundary
// will be rendered in place of the search bar in the initial HTML.
// When the value is available during React hydration the fallback
// will be replaced with the `<SearchBar>` component.
function SearchBarFallback() {
  return <>placeholder</>
}

export default function Page() {
  return (
    <>
      <nav>
        <Suspense fallback={<SearchBarFallback />}>
          <SearchBar />
        </Suspense>
      </nav>
      <h1>Dashboard</h1>
    </>
  )
}
```

### Dynamic Rendering

If a route is [dynamically rendered](/docs/app/building-your-application/rendering/server-components#dynamic-rendering), `useSearchParams` will be available on the server during the initial server render of the Client Component.

> **Good to know**: Setting the [`dynamic` route segment config option](/docs/app/api-reference/file-conventions/route-segment-config#dynamic) to `force-dynamic` can be used to force dynamic rendering.

For example:

```tsx filename="app/dashboard/search-bar.tsx" switcher
'use client'
```

~~~
import { useSearchParams } from 'next/navigation'

export default function SearchBar() {
  const searchParams = useSearchParams()

  const search = searchParams.get('search')

  // This will be logged on the server during the initial render
  // and on the client on subsequent navigations.
  console.log(search)

  return <>Search: {search}</>
}
```

```jsx filename="app/dashboard/search-bar.js" switcher
'use client'

import { useSearchParams } from 'next/navigation'

export default function SearchBar() {
  const searchParams = useSearchParams()

  const search = searchParams.get('search')

  // This will be logged on the server during the initial render
  // and on the client on subsequent navigations.
  console.log(search)

  return <>Search: {search}</>
}
```

```tsx filename="app/dashboard/page.tsx" switcher
import SearchBar from './search-bar'

export const dynamic = 'force-dynamic'

export default function Page() {
  return (
    <>
      <nav>
        <SearchBar />
      </nav>
      <h1>Dashboard</h1>
    </>
~~~

)
}

```jsx filename="app/dashboard/page.js" switcher
import SearchBar from './search-bar'

export const dynamic = 'force-dynamic'

export default function Page() {
  return (
    <>
      <nav>
        <SearchBar />
      </nav>
      <h1>Dashboard</h1>
    </>
  )
}
```

### Server Components

#### Pages

To access search params in [Pages](/docs/app/api-reference/file-conventions/page) (Server Components), use the [`searchParams`](/docs/app/api-reference/file-conventions/page#searchparams-optional) prop.

#### Layouts

Unlike Pages, [Layouts](/docs/app/api-reference/file-conventions/layout) (Server Components) **do not** receive the `searchParams` prop. This is because a shared layout is [not re-rendered during navigation](/docs/app/building-your-application/routing/linking-and-navigating#3-partial-rendering) which could lead to stale `searchParams` between navigations. View [detailed explanation](/docs/app/api-reference/file-conventions/layout#layouts-do-not-receive-searchparams).

Instead, use the Page [`searchParams`](/docs/app/api-reference/file-conventions/page) prop or the [`useSearchParams`](/docs/app/api-reference/functions/use-search-params) hook in a Client Component, which is re-rendered on the client with the latest `searchParams`.

## Examples

### Updating `searchParams`

You can use [`useRouter`](/docs/app/api-reference/functions/use-router) or [`Link`](/docs/app/api-reference/components/link) to set new `searchParams`. After a navigation is performed, the current [`page.js`](/docs/app/api-reference/file-conventions/page) will receive an updated [`searchParams` prop](/docs/app/api-reference/file-conventions/page#searchparams-optional).

```tsx filename="app/example-client-component.tsx" switcher
export default function ExampleClientComponent() {
  const router = useRouter()
  const pathname = usePathname()
  const searchParams = useSearchParams()!

  // Get a new searchParams string by merging the current
  // searchParams with a provided key/value pair
  const createQueryString = useCallback(
    (name: string, value: string) => {
      const params = new URLSearchParams(searchParams)
      params.set(name, value)

      return params.toString()
    },
    [searchParams]
  )

  return (
    <>
      <p>Sort By</p>

      {/* using useRouter */}
      <button
        onClick={() => {
          // <pathname>?sort=asc
          router.push(pathname + '?' + createQueryString('sort', 'asc'))
        }}
      >
        ASC
      </button>

      {/* using <Link> */}
      <Link
        href={
          // <pathname>?sort=desc
          pathname + '?' + createQueryString('sort', 'desc')
        }
      >
        DESC
```

      </Link>
    </>
)
}
~~~

```jsx filename="app/example-client-component.js" switcher
export default function ExampleClientComponent() {
  const router = useRouter()
  const pathname = usePathname()
  const searchParams = useSearchParams()

  // Get a new searchParams string by merging the current
  // searchParams with a provided key/value pair
  const createQueryString = useCallback(
    (name, value) => {
      const params = new URLSearchParams(searchParams)
      params.set(name, value)

      return params.toString()
    },
    [searchParams]
  )

  return (
    <>
      <p>Sort By</p>

      {/* using useRouter */}
      <button
        onClick={() => {
          // <pathname>?sort=asc
          router.push(pathname + '?' + createQueryString('sort', 'asc'))
        }}
      >
        ASC
      </button>

      {/* using <Link> */}
      <Link
        href={
          // <pathname>?sort=desc
          pathname + '?' + createQueryString('sort', 'desc')
        }
      >
        DESC
      </Link>
~~~

    </>
  )
}

## Version History

| Version   | Changes                       |
| --------- | ----------------------------- |
| `v13.0.0` | `useSearchParams` introduced. |

---
title: useSelectedLayoutSegment
description: API Reference for the useSelectedLayoutSegment hook.
---

`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.

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

export default function ExampleClientComponent() {
  const segment = useSelectedLayoutSegment()

  return <p>Active segment: {segment}</p>
}
```

```jsx filename="app/example-client-component.js" switcher
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

export default function ExampleClientComponent() {
  const segment = useSelectedLayoutSegment()

  return <p>Active segment: {segment}</p>
}
```

> **Good to know**:

>
> - Since `useSelectedLayoutSegment` is a [Client Component](/docs/app/building-your-application/rendering/client-components) hook, and Layouts are [Server Components](/docs/app/building-your-application/rendering/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`](/docs/app/api-reference/functions/use-selected-layout-segments)

## Parameters

```tsx
const segment = useSelectedLayoutSegment(parallelRoutesKey?: string)
```

`useSelectedLayoutSegment` _optionally_ accepts a [`parallelRoutesKey`](/docs/app/building-your-application/routing/parallel-routes#useselectedlayoutsegments), which allows you to read the active route segment within that slot.

## 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`	`/`	`null`
`app/layout.js`	`/dashboard`	`'dashboard'`
`app/dashboard/layout.js`	`/dashboard`	`null`
`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:

```tsx filename="app/blog/blog-nav-link.tsx" switcher
```

~~~
'use client'

import Link from 'next/link'
import { useSelectedLayoutSegment } from 'next/navigation'

// This *client* component will be imported into a blog layout
export default function BlogNavLink({
  slug,
  children,
}: {
  slug: string
  children: React.ReactNode
}) {
  // Navigating to `/blog/hello-world` will return 'hello-world'
  // for the selected layout segment
  const segment = useSelectedLayoutSegment()
  const isActive = slug === segment

  return (
    <Link
      href={`/blog/${slug}`}
      // Change style depending on whether the link is active
      style={{ fontWeight: isActive ? 'bold' : 'normal' }}
    >
      {children}
    </Link>
  )
}
```

```jsx filename="app/blog/blog-nav-link.js" switcher
'use client'

import Link from 'next/link'
import { useSelectedLayoutSegment } from 'next/navigation'

// This *client* component will be imported into a blog layout
export default function BlogNavLink({ slug, children }) {
  // Navigating to `/blog/hello-world` will return 'hello-world'
  // for the selected layout segment
  const segment = useSelectedLayoutSegment()
  const isActive = slug === segment

  return (
    <Link
      href={`/blog/${slug}`}
      // Change style depending on whether the link is active
~~~

```
   style={{ fontWeight: isActive ? 'bold' : 'normal' }}
  >
   {children}
  </Link>
 )
}
```

```tsx filename="app/blog/layout.tsx" switcher
// Import the Client Component into a parent Layout (Server Component)
import { BlogNavLink } from './blog-nav-link'
import getFeaturedPosts from './get-featured-posts'

export default async function Layout({
 children,
}: {
 children: React.ReactNode
}) {
 const featuredPosts = await getFeaturedPosts()
 return (
  <div>
   {featuredPosts.map((post) => (
    <div key={post.id}>
     <BlogNavLink slug={post.slug}>{post.title}</BlogNavLink>
    </div>
   ))}
   <div>{children}</div>
  </div>
 )
}
```

```jsx filename="app/blog/layout.js" switcher
// Import the Client Component into a parent Layout (Server Component)
import { BlogNavLink } from './blog-nav-link'
import getFeaturedPosts from './get-featured-posts'

export default async function Layout({ children }) {
 const featuredPosts = await getFeaturedPosts()
 return (
  <div>
   {featuredPosts.map((post) => (
    <div key={post.id}>
     <BlogNavLink slug={post.slug}>{post.title}</BlogNavLink>
    </div>
   ))}
   <div>{children}</div>
```

```jsx
    </div>
  )
}
```

## Version History

Version	Changes
`v13.0.0`	`useSelectedLayoutSegment` introduced.

---
title: useSelectedLayoutSegments
description: API Reference for the useSelectedLayoutSegments hook.

`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.

```tsx filename="app/example-client-component.tsx" switcher
'use client'

import { useSelectedLayoutSegments } from 'next/navigation'

export default function ExampleClientComponent() {
  const segments = useSelectedLayoutSegments()

  return (
    <ul>
      {segments.map((segment, index) => (
        <li key={index}>{segment}</li>
      ))}
    </ul>
  )
}
```

```jsx filename="app/example-client-component.js" switcher
'use client'

import { useSelectedLayoutSegments } from 'next/navigation'

export default function ExampleClientComponent() {
  const segments = useSelectedLayoutSegments()
  return (
    <ul>
      {segments.map((segment, index) => (
        <li key={index}>{segment}</li>
      ))}
    </ul>
  )
}
```

> **Good to know**:
>
> - Since `useSelectedLayoutSegments` is a [Client Component](/docs/app/building-your-application/rendering/client-components) hook, and Layouts are [Server Components](/docs/app/building-your-application/rendering/server-components) by default, `useSelectedLayoutSegments` is usually called via a Client Component that is imported into a Layout.
> - The returned segments include [Route Groups](/docs/app/building-your-application/routing/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

```tsx
const segments = useSelectedLayoutSegments(parallelRoutesKey?: string)
```

`useSelectedLayoutSegments` _optionally_ accepts a [`parallelRoutesKey`](/docs/app/building-your-application/routing/parallel-routes#useselectedlayoutsegments), which allows you to read the active route segment within that slot.

## 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']`

## Version History

Version	Changes
`v13.0.0`	`useSelectedLayoutSegments` introduced.

---
title: appDir
description: Enable the App Router to use layouts, streaming, and more.

> **Good to know**: This option is **no longer** needed as of Next.js 13.4. The App Router is now stable.

The App Router ([`app` directory](/docs/app/building-your-application/routing)) enables support for [layouts](/docs/app/building-your-application/routing/pages-and-layouts), [Server Components](/docs/app/building-your-application/rendering/server-components), [streaming](/docs/app/building-your-application/routing/loading-ui-and-streaming), and [colocated data fetching](/docs/app/building-your-application/data-fetching).

Using the `app` directory will automatically enable [React Strict Mode](https://react.dev/reference/react/StrictMode). Learn how to [incrementally adopt `app`](/docs/app/building-your-application/upgrading/app-router-migration#migrating-from-pages-to-app).

</AppOnly>

<PagesOnly>

> **Attention**: [Deploying to Vercel](/docs/pages/building-your-application/deploying) automatically configures a global CDN for your Next.js project.
> You do not need to manually setup an Asset Prefix.

</PagesOnly>

> **Good to know**: Next.js 9.5+ added support for a customizable [Base Path](/docs/app/api-reference/next-config-js/basePath), 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](https://en.wikipedia.org/wiki/Content_delivery_network), 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 filename="next.config.js"
const isProd = process.env.NODE_ENV === 'production'

module.exports = {
  // Use the CDN in production and localhost for development.
  assetPrefix: isProd ? 'https://cdn.mydomain.com' : undefined,
}
```

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.4d708494b3aed70c04f0.js
```

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:

<AppOnly>

- Files in the [public](/docs/app/building-your-application/optimizing/static-assets) folder; if you want to serve those assets over a CDN, you'll have to introduce the prefix yourself

</AppOnly>

<PagesOnly>

- Files in the [public](/docs/pages/building-your-application/optimizing/static-assets) 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](/docs/pages/building-your-application/data-fetching/incremental-static-regeneration), even if you're not using it (for consistency).

</PagesOnly>

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  basePath: '/docs',
}
```

> **Good to know**: 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`.

```js
export default function HomePage() {
  return (
    <>
      <Link href="/about">About Page</Link>
    </>
  )
}
```

Output html:

```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

<AppOnly>

When using the [`next/image`](/docs/app/api-reference/components/image) component, you will need to add the `basePath` in front of `src`.

</AppOnly>

<PagesOnly>

When using the [`next/image`](/docs/pages/api-reference/components/image) component, you will need to add the `basePath` in front of `src`.

</PagesOnly>

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

```jsx
import Image from 'next/image'

function Home() {
  return (
    <>
      <h1>My Homepage</h1>
      <Image
        src="/docs/me.png"
        alt="Picture of the author"
        width={500}
        height={500}
      />
      <p>Welcome to my homepage!</p>
    </>
  )
}

export default Home
```

---
title: compress
description: Next.js provides gzip compression to compress rendered content and static files, it only works with the server target. Learn more about it here.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js provides [**gzip**](https://tools.ietf.org/html/rfc6713#section-3) compression to compress rendered content and static files. In general you will want to enable compression on a HTTP proxy like [nginx](https://www.nginx.com/), to offload load from the `Node.js` process.

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

```js filename="next.config.js"
module.exports = {
  compress: false,
}
```
---
title: devIndicators
description: Optimized pages include an indicator to let you know if it's being statically optimized. You can opt-out of it here.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<AppOnly>

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

> **Good to know**: 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 filename="next.config.js"
module.exports = {
  devIndicators: {
    buildActivityPosition: 'bottom-right',
  },
}
```

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 filename="next.config.js"
module.exports = {
  devIndicators: {
    buildActivity: false,
  },
}
```

</AppOnly>

<PagesOnly>

> **Good to know**: This indicator was removed in Next.js version 10.0.1. We recommend upgrading to the latest version of Next.js.

When a page qualifies for [Automatic Static Optimization](/docs/pages/building-your-application/rendering/automatic-static-optimization) we show an indicator to let you know.

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

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

```js filename="next.config.js"
module.exports = {
  devIndicators: {
    autoPrerender: false,
  },
}
```

</PagesOnly>
---
title: distDir
description: Set a custom build directory to use instead of the default .next directory.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  distDir: 'build',
}
```

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.

---
title: env

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<AppOnly>

> Since the release of [Next.js 9.4](https://nextjs.org/blog/next-9-4) we now have a more intuitive and ergonomic experience for [adding environment variables](/docs/app/building-your-application/configuring/environment-variables). Give it a try!

</AppOnly>

<PagesOnly>

> Since the release of [Next.js 9.4](https://nextjs.org/blog/next-9-4) we now have a more intuitive and ergonomic experience for [adding environment variables](/docs/pages/building-your-application/configuring/environment-variables). Give it a try!

</PagesOnly>

<details>
  <summary>Examples</summary>

- [With env](https://github.com/vercel/next.js/tree/canary/examples/with-env-from-next-config-js)

</details>

<AppOnly>

> **Good to know**: 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](/docs/app/building-your-application/configuring/environment-variables).

</AppOnly>

<PagesOnly>

> **Good to know**: 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](/docs/pages/building-your-application/configuring/environment-variables).

</PagesOnly>

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

```js filename="next.config.js"
module.exports = {
  env: {
    customKey: 'my-value',
  },
}
```

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

```jsx
function Page() {
  return <h1>The value of customKey is: {process.env.customKey}</h1>
}

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](https://webpack.js.org/plugins/define-plugin/).

For example, the following line:

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

Will end up being:

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

---

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  eslint: {
    // Warning: This allows production builds to successfully complete even if
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
}
```

---
title: exportPathMap (Deprecated)
nav_title: exportPathMap
description: Customize the pages that will be exported as HTML files when using `next export`.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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

<details>

  <summary>Examples</summary>

- [Static Export](https://github.com/vercel/next.js/tree/canary/examples/with-static-export)

</details>

`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`](/docs/app/api-reference/next-cli#development).

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 filename="next.config.js"
module.exports = {
  exportPathMap: async function (
    defaultPathMap,
    { dev, dir, outDir, distDir, buildId }
  ) {
    return {
      '/': { page: '/' },
      '/about': { page: '/about' },
      '/p/hello-nextjs': { page: '/post', query: { title: 'hello-nextjs' } },
      '/p/learn-nextjs': { page: '/post', query: { title: 'learn-nextjs' } },
      '/p/deploy-nextjs': { page: '/post', query: { title: 'deploy-nextjs' } },
    }
  },
}
```

> **Good to know**: the `query` field in `exportPathMap` cannot be used with [automatically statically optimized pages](/docs/pages/building-your-application/rendering/automatic-static-optimization) or [`getStaticProps` pages](/docs/pages/building-your-application/data-fetching/get-static-props) 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`](#customizing-the-output-directory)). When `dev` is `true` the value of `outDir` will be `null`.
- `distDir` - Absolute path to the `.next/` directory (configurable with the [`distDir`](/docs/pages/api-reference/next-config-js/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 filename="next.config.js"
module.exports = {
  trailingSlash: true,
}
```

## Customizing the output directory

<AppOnly>

[`next export`](/docs/app/building-your-application/deploying/static-exports) will use `out` as the default output directory, you can customize this using the `-o` argument, like so:

</AppOnly>

<PagesOnly>

[`next export`](/docs/pages/building-your-application/deploying/static-exports) will use `out` as the default output directory, you can customize this using the `-o` argument, like so:

</PagesOnly>

```bash filename="Terminal"
next export -o outdir
```

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

---
title: generateBuildId
description: Configure the build id, which is used to identify the current build in which your application is being served.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js generates an ID during `next build` to identify which version of your application is being served. The same build should be used and boot up multiple containers.

If you are rebuilding for each stage of your environment, you will need to generate a consistent build ID to use between containers. Use the `generateBuildId` command in `next.config.js`:

```jsx filename="next.config.js"
module.exports = {
  generateBuildId: async () => {
    // This could be anything, using the latest git hash
    return process.env.GIT_HASH
  },
}
```

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js will generate [etags](https://en.wikipedia.org/wiki/HTTP_ETag) 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 filename="next.config.js"
module.exports = {
  generateEtags: false,
}
```

---
title: headers
description: Add custom HTTP headers to your Next.js app.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  async headers() {
    return [
      {
        source: '/about',
        headers: [
          {
            key: 'x-custom-header',
            value: 'my custom header value',
          },
          {
            key: 'x-another-custom-header',
            value: 'my other custom header value',
          },
        ],
      },
    ]
  },
}
```

`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](#header-cookie-and-query-matching) with the `type`, `key` and `value` properties.
- `missing` is an array of [missing objects](#header-cookie-and-query-matching) 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 filename="next.config.js"
module.exports = {
  async headers() {
    return [
      {
        source: '/:path*',
        headers: [
          {
            key: 'x-hello',
```

            value: 'there',
          },
        ],
      },
      {
        source: '/hello',
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
    ]
  },
}

## Path Matching

Path matches are allowed, for example `/blog/:slug` will match `/blog/hello-world` (no nested paths):

```js filename="next.config.js"
module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:slug',
        headers: [
          {
            key: 'x-slug',
            value: ':slug', // Matched parameters can be used in the value
          },
          {
            key: 'x-slug-:slug', // Matched parameters can be used in the key
            value: 'my other custom header value',
          },
        ],
      },
    ]
  },
}
```

### 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 filename="next.config.js"
module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:slug*',
        headers: [
          {
            key: 'x-slug',
            value: ':slug*', // Matched parameters can be used in the value
          },
          {
            key: 'x-slug-:slug*', // Matched parameters can be used in the key
            value: 'my other custom header value',
          },
        ],
      },
    ]
  },
}
```

### 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 filename="next.config.js"
module.exports = {
  async headers() {
    return [
      {
        source: '/blog/:post(||d{1,})',
        headers: [
          {
            key: 'x-post',
            value: ':post',
          },
        ],
      },
    ]
  },
}
```

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 filename="next.config.js"
module.exports = {
  async headers() {
    return [
      {
        // this will match `/english(default)/something` being requested
        source: '/english\\(default\\)/:slug',
        headers: [
          {
            key: 'x-header',
            value: 'value',
          },
        ],
      },
    ]
  },
}
```

## 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 filename="next.config.js"
module.exports = {
  async headers() {
    return [
      // if the header `x-add-header` is present,
      // the `x-another-header` header will be applied
```

```
{
  source: '/:path*',
  has: [
    {
      type: 'header',
      key: 'x-add-header',
    },
  ],
  headers: [
    {
      key: 'x-another-header',
      value: 'hello',
    },
  ],
},
// if the header `x-no-header` is not present,
// the `x-another-header` header will be applied
{
  source: '/:path*',
  missing: [
    {
      type: 'header',
      key: 'x-no-header',
    },
  ],
  headers: [
    {
      key: 'x-another-header',
      value: 'hello',
    },
  ],
},
// if the source, query, and cookie are matched,
// the `x-authorized` header will be applied
{
  source: '/specific/:path*',
  has: [
    {
      type: 'query',
      key: 'page',
      // the page value will not be available in the
      // header key/values since value is provided and
      // doesn't use a named capture group e.g. (?<page>home)
      value: 'home',
    },
    {
      type: 'cookie',
        key: 'authorized',
        value: 'true',
      },
    ],
    headers: [
      {
        key: 'x-authorized',
        value: ':authorized',
      },
    ],
  },
  // if the header `x-authorized` is present and
  // contains a matching value, the `x-another-header` will be applied
  {
    source: '/:path*',
    has: [
      {
        type: 'header',
        key: 'x-authorized',
        value: '(?<authorized>yes|true)',
      },
    ],
    headers: [
      {
        key: 'x-another-header',
        value: ':authorized',
      },
    ],
  },
  // if the host is `example.com`,
  // this header will be applied
  {
    source: '/:path*',
    has: [
      {
        type: 'host',
        value: 'example.com',
      },
    ],
    headers: [
      {
        key: 'x-another-header',
        value: ':authorized',
      },
    ],
  },
]
```

  },
}

## Headers with basePath support

When leveraging [`basePath` support](/docs/app/api-reference/next-config-js/basePath) with headers each `source` is automatically prefixed with the `basePath` unless you add `basePath: false` to the header:

```js filename="next.config.js"
module.exports = {
  basePath: '/docs',

  async headers() {
    return [
      {
        source: '/with-basePath', // becomes /docs/with-basePath
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        source: '/without-basePath', // is not modified since basePath: false is set
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
        basePath: false,
      },
    ]
  },
}
```

## Headers with i18n support

<AppOnly>

When leveraging [`i18n` support](/docs/app/building-your-application/routing/internationalization) 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.

</AppOnly>

<PagesOnly>

When leveraging [`i18n` support](/docs/pages/building-your-application/routing/internationalization) 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.

</PagesOnly>

```js filename="next.config.js"
module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },

  async headers() {
    return [
      {
        source: '/with-locale', // automatically handles all locales
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // does not handle locales automatically since locale: false is set
        source: '/nl/with-locale-manual',
        locale: false,
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // this matches '/' since `en` is the defaultLocale
        source: '/en',
        locale: false,
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
      {
        // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
        // `/` or `/fr` routes like /:path* would
        source: '/(.*)',
        headers: [
          {
            key: 'x-hello',
            value: 'world',
          },
        ],
      },
    ]
  },
}
```

## Cache-Control

You cannot set `Cache-Control` headers in `next.config.js` for pages or assets, as these headers will be overwritten in production to ensure that responses and static assets are cached effectively.

<AppOnly>

Learn more about [caching](/docs/app/building-your-application/caching) with the App Router.

</AppOnly>

<PagesOnly>

If you need to revalidate the cache of a page that has been [statically generated](/docs/pages/building-your-application/rendering/static-site-generation), you can do so by setting the `revalidate` prop in the page's [`getStaticProps`](/docs/pages/building-your-application/data-fetching/get-static-props) function.

You can set the `Cache-Control` header in your [API Routes](/docs/pages/building-your-application/routing/api-routes) by using the `res.setHeader` method:

```ts filename="pages/api/hello.ts" switcher
import type { NextApiRequest, NextApiResponse } from 'next'

type ResponseData = {
  message: string
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>
) {
  res.setHeader('Cache-Control', 's-maxage=86400')
  res.status(200).json({ message: 'Hello from Next.js!' })
}
```

```js filename="pages/api/hello.js" switcher
export default function handler(req, res) {
  res.setHeader('Cache-Control', 's-maxage=86400')
  res.status(200).json({ message: 'Hello from Next.js!' })
}
```

</PagesOnly>

## Options

### X-DNS-Prefetch-Control

[This header](https://developer.mozilla.org/docs/Web/HTTP/Headers/X-DNS-Prefetch-Control) 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](https://developer.mozilla.org/docs/Glossary/DNS) is more likely to be resolved by the time the referenced items are needed. This reduces latency when the user clicks a link.

```js
{
  key: 'X-DNS-Prefetch-Control',
  value: 'on'
}
```

### Strict-Transport-Security

[This header](https://developer.mozilla.org/docs/Web/HTTP/Headers/Strict-Transport-Security) 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](https://vercel.com/docs/concepts/edge-network/headers#strict-transport-security?utm_source=next-site&utm_medium=docs&utm_campaign=next-website), this header is not necessary as it's automatically added to all deployments unless you declare `headers` in your `next.config.js`.

```js
{
  key: 'Strict-Transport-Security',
  value: 'max-age=63072000; includeSubDomains; preload'
}
```

### X-Frame-Options

[This header](https://developer.mozilla.org/docs/Web/HTTP/Headers/X-Frame-Options) 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.

```js
{
  key: 'X-Frame-Options',
  value: 'SAMEORIGIN'
}
```

### Permissions-Policy

[This header](https://developer.mozilla.org/docs/Web/HTTP/Headers/Permissions-Policy) allows you to control which features and APIs can be used in the browser. It was previously named `Feature-Policy`.

```js
{
  key: 'Permissions-Policy',
  value: 'camera=(), microphone=(), geolocation=(), browsing-topics=()'
}
```

### X-Content-Type-Options

[This header](https://developer.mozilla.org/docs/Web/HTTP/Headers/X-Content-Type-Options) 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`.

```js
{
  key: 'X-Content-Type-Options',
  value: 'nosniff'
}
```

### Referrer-Policy

[This header](https://developer.mozilla.org/docs/Web/HTTP/Headers/Referrer-Policy) controls how much information the browser includes when navigating from the current website (origin) to another.

```js
{
  key: 'Referrer-Policy',
  value: 'origin-when-cross-origin'
}
```

### Content-Security-Policy

Learn more about adding a [Content Security Policy](/docs/app/building-your-application/configuring/content-security-policy) to your application.

## Version History

Version	Changes
`v13.3.0`	`missing` added.
`v10.2.0`	`has` added.
`v9.5.0`	Headers added.

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

In Node.js versions prior to 18, Next.js automatically polyfills `fetch()` with [undici](/docs/architecture/supported-browsers#polyfills) and enables [HTTP Keep-Alive](https://developer.mozilla.org/docs/Web/HTTP/Headers/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 filename="next.config.js"
module.exports = {
  httpAgentOptions: {
    keepAlive: false,
  },
}
```

---
title: images
description: Custom configuration for the next/image loader

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
```

  },
}

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:

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

<AppOnly>

Alternatively, you can use the [`loader` prop](/docs/app/api-reference/components/image#loader) to pass the function to each instance of `next/image`.

</AppOnly>

<PagesOnly>

Alternatively, you can use the [`loader` prop](/docs/pages/api-reference/components/image#loader) to pass the function to each instance of `next/image`.

</PagesOnly>

## Example Loader Configuration

- [Akamai](#akamai)
- [Cloudinary](#cloudinary)
- [Cloudflare](#cloudflare)
- [Contentful](#contentful)
- [Fastly](#fastly)
- [Gumlet](#gumlet)
- [ImageEngine](#imageengine)
- [Imgix](#imgix)
- [Thumbor](#thumbor)
- [Sanity](#sanity)
- [Supabase](#supabase)

### Akamai

```js
```

// Docs: https://techdocs.akamai.com/ivm/reference/test-images-on-demand
export default function akamaiLoader({ src, width, quality }) {
  return `https://example.com/${src}?imwidth=${width}`
}

### Cloudinary

```js
// Demo: https://res.cloudinary.com/demo/image/upload/w_300,c_limit,q_auto/
turtles.jpg
export default function cloudinaryLoader({ src, width, quality }) {
  const params = ['f_auto', 'c_limit', `w_${width}`, `q_${quality || 'auto'}`]
  return `https://example.com/${params.join(',')}${src}`
}
```

### Cloudflare

```js
// Docs: https://developers.cloudflare.com/images/url-format
export default function cloudflareLoader({ src, width, quality }) {
  const params = [`width=${width}`, `quality=${quality || 75}`, 'format=auto']
  return `https://example.com/cdn-cgi/image/${params.join(',')}/${src}`
}
```

### Contentful

```js
// Docs: https://www.contentful.com/developers/docs/references/images-api/
export default function contentfulLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('fm', 'webp')
  url.searchParams.set('w', width.toString())
  url.searchParams.set('q', (quality || 75).toString())
  return url.href
}
```

### Fastly

```js
// Docs: https://developer.fastly.com/reference/io/
export default function fastlyLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('auto', 'webp')
```

  url.searchParams.set('width', width.toString())
  url.searchParams.set('quality', (quality || 75).toString())
  return url.href
}

### Gumlet

```js
// Docs: https://docs.gumlet.com/reference/image-transform-size
export default function gumletLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('format', 'auto')
  url.searchParams.set('w', width.toString())
  url.searchParams.set('q', (quality || 75).toString())
  return url.href
}
```

### ImageEngine

```js
// Docs: https://support.imageengine.io/hc/en-us/articles/360058880672-Directives
export default function imageengineLoader({ src, width, quality }) {
  const compression = 100 - (quality || 50)
  const params = [`w_${width}`, `cmpr_${compression}`)]
  return `https://example.com${src}?imgeng=/${params.join('/')`
}
```

### Imgix

```js
// Demo: https://static.imgix.net/daisy.png?format=auto&fit=max&w=300
export default function imgixLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  const params = url.searchParams
  params.set('auto', params.getAll('auto').join(',') || 'format')
  params.set('fit', params.get('fit') || 'max')
  params.set('w', params.get('w') || width.toString())
  params.set('q', (quality || 50).toString())
  return url.href
}
```

### Thumbor

```js
// Docs: https://thumbor.readthedocs.io/en/latest/
export default function thumborLoader({ src, width, quality }) {
  const params = [`${width}x0`, `filters:quality(${quality || 75})`]
  return `https://example.com${params.join('/')}${src}`
}
```

### Sanity

```js
// Docs: https://www.sanity.io/docs/image-urls
export default function sanityLoader({ src, width, quality }) {
  const prj = 'zp7mbokg'
  const dataset = 'production'
  const url = new URL(`https://cdn.sanity.io/images/${prj}/${dataset}${src}`)
  url.searchParams.set('auto', 'format')
  url.searchParams.set('fit', 'max')
  url.searchParams.set('w', width.toString())
  if (quality) {
    url.searchParams.set('q', quality.toString())
  }
  return url.href
}
```

### Supabase

```js
// Docs: https://supabase.com/docs/guides/storage/image-transformations#nextjs-loader
export default function supabaseLoader({ src, width, quality }) {
  const url = new URL(`https://example.com${src}`)
  url.searchParams.set('width', width.toString())
  url.searchParams.set('quality', (quality || 75).toString())
  return url.href
}
```

---
title: incrementalCacheHandlerPath
description: Configure the Next.js cache used for storing and revalidating data.

In Next.js, the [default cache handler](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating) 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 filename="next.config.js"
module.exports = {
  experimental: {
    incrementalCacheHandlerPath: require.resolve('./cache-handler.js'),
  },
}
```

Here's an example of a custom cache handler:

```js filename="cache-handler.js"
const cache = new Map()

module.exports = class CacheHandler {
  constructor(options) {
    this.options = options
    this.cache = {}
  }

  async get(key) {
    return cache.get(key)
  }

  async set(key, data) {
    cache.set(key, {
      value: data,
      lastModified: Date.now(),
    })
  }
}
```

## API Reference

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

### `get()`

Parameter	Type	Description
`key`	`string`	The key to the cached value.

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

### `set()`

Parameter	Type	Description
`key`	`string`	The key to store the data under.
`data`	Data or `null`	The data to be cached.

Returns `Promise<void>`.

### `revalidateTag()`

Parameter	Type	Description
`tag`	`string`	The cache tag to revalidate.

Returns `Promise<void>`. Learn more about [revalidating data](/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating) or the [`revalidateTag()`](/docs/app/api-reference/functions/revalidateTag) function.

---
title: next.config.js Options
description: Learn how to configure your application with next.config.js.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Next.js can be configured through a `next.config.js` file in the root of your project directory (for example, by `package.json`).

```js filename="next.config.js"
/** @type {import('next').NextConfig} */
const nextConfig = {
  /* config options here */
}

module.exports = nextConfig
```

`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.

If you need [ECMAScript modules](https://nodejs.org/api/esm.html), you can use `next.config.mjs`:

```js filename="next.config.mjs"
/**
 * @type {import('next').NextConfig}
 */
const nextConfig = {
  /* config options here */
}

export default nextConfig
```

You can also use a function:

```js filename="next.config.mjs"
export default (phase, { defaultConfig }) => {
  /**
   * @type {import('next').NextConfig}
   */
  const nextConfig = {
    /* config options here */
  }
  return nextConfig
}
```

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

```js filename="next.config.js"
module.exports = async (phase, { defaultConfig }) => {
  /**
   * @type {import('next').NextConfig}
   */
  const nextConfig = {
    /* config options here */
  }
  return nextConfig
}
```

`phase` is the current context in which the configuration is loaded. You can see the [available phases](https://github.com/vercel/next.js/blob/5e6b008b561caf2710ab7be63320a3d549474a5b/packages/next/shared/lib/constants.ts#L19-L23). Phases can be imported from `next/constants`:

```js
```

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

module.exports = (phase, { defaultConfig }) => {
  if (phase === PHASE_DEVELOPMENT_SERVER) {
    return {
      /* development only config options here */
    }
  }

  return {
    /* config options for all phases except development here */
  }
}
```

The commented lines are the place where you can put the configs allowed by `next.config.js`, which are [defined in this file](https://github.com/vercel/next.js/blob/canary/packages/next/src/server/config-shared.ts).

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.

This page documents all the available configuration options:

---
title: logging
description: Configure how data fetches are logged to the console when running Next.js in development mode.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

You can configure the logging level and whether the full URL is logged to the console when running Next.js in development mode.

Currently, `logging` only applies to data fetching using the `fetch` API. It does not yet apply to other logs inside of Next.js.

```js filename="next.config.js"
module.exports = {
  logging: {
    fetches: {
      fullUrl: true,
    },
  },
}
```

---
title: mdxRs
description: Use the new Rust compiler to compile MDX files in the App Router.

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

```js filename="next.config.js"
const withMDX = require('@next/mdx')()

/** @type {import('next').NextConfig} */
const nextConfig = {
  pageExtensions: ['ts', 'tsx', 'mdx'],
  experimental: {
    mdxRs: true,
  },
}

module.exports = withMDX(nextConfig)
```

---
title: onDemandEntries
description: Configure how Next.js will dispose and keep in memory pages created in development.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
```

module.exports = {
  onDemandEntries: {
    // period (in ms) where the server will keep pages in the buffer
    maxInactiveAge: 25 * 1000,
    // number of pages that should be kept simultaneously without being
disposed
    pagesBufferLength: 2,
  },
}

---
title: optimizePackageImports
description: API Reference for optmizedPackageImports Next.js Config Option
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Some packages can export hundreds or thousands of modules, which can cause performance issues in development and production.

Adding a package to `experimental.optimizePackageImports` will only load the modules you are actually using, while still giving you the convenience of writing import statements with many named exports.

```js filename="next.config.js"
module.exports = {
  experimental: {
    optimizePackageImports: ['package-name'],
  },
}
```

Libraries like `@mui/icons-material`, `@mui/material`, `date-fns`, `lodash`, `lodash-es`, `react-bootstrap`, `@headlessui/react`, `@heroicons/react`, and `lucide-react` are already optimized by default.

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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`](https://github.com/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`:

```js filename="next.config.js"
module.exports = {
  output: 'standalone',
}
```

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.

<AppOnly>

> **Good to know**:
>
> - If your project needs to listen to a specific port or hostname, you can define `PORT` or `HOSTNAME` environment variables before running `server.js`. For example, run `PORT=8080 HOSTNAME=0.0.0.0 node server.js` to start the server on `http://0.0.0.0:8080`.
> - If your project uses [Image Optimization](/docs/app/building-your-application/optimizing/images) with the default `loader`, you must install `sharp` as a dependency:

</AppOnly>

<PagesOnly>

> **Good to know**:
>
> - `next.config.js` is read during `next build` and serialized into the `server.js` output file. If the legacy [`serverRuntimeConfig` or `publicRuntimeConfig` options](/docs/pages/api-reference/next-config-js/runtime-configuration) are being used, the values will be specific to values at build time.
> - If your project needs to listen to a specific port or hostname, you can define `PORT` or `HOSTNAME` environment variables before running `server.js`. For example, run `PORT=8080 HOSTNAME=0.0.0.0 node server.js` to start the server on `http://0.0.0.0:8080`.
> - If your project uses [Image Optimization](/docs/pages/building-your-application/optimizing/images) with the default `loader`, you must install `sharp` as a dependency:

</PagesOnly>

```bash filename="Terminal"
npm i sharp
```

```bash filename="Terminal"
yarn add sharp
```

```bash filename="Terminal"
pnpm add sharp
```

```bash filename="Terminal"
bun 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 filename="packages/web-app/next.config.js"
module.exports = {
  experimental: {
    // this includes files from the monorepo base two directories up
    outputFileTracingRoot: path.join(__dirname, '../../'),
  },
}
```

- 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](https://www.npmjs.com/package/minimatch) 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 filename="next.config.js"
module.exports = {
  experimental: {
    outputFileTracingExcludes: {
      '/api/hello': ['./un-necessary-folder/**/*'],
    },
    outputFileTracingIncludes: {
      '/api/another': ['./necessary-folder/**/*'],
    },
  },
}
```

- 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](https://vercel.com), 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`:

```js filename="next.config.js"
module.exports = {
  experimental: {
    turbotrace: {
      // control the log level of the turbotrace, default is `error`
      logLevel?:
      | 'bug'
      | 'fatal'
      | 'error'
      | 'warning'
      | 'hint'
      | 'note'
      | 'suggestions'
      | 'info',
      // control if the log of turbotrace should contain the details of the analysis,
default is `false`
      logDetail?: boolean
      // show all log messages without limit
      // turbotrace only show 1 log message for each categories by default
      logAll?: boolean
      // control the context directory of the turbotrace
      // files outside of the context directory will not be traced
      // set the `experimental.outputFileTracingRoot` has the same effect
      // if the `experimental.outputFileTracingRoot` and this option are both set,
the `experimental.turbotrace.contextDirectory` will be used
      contextDirectory?: string
      // if there is `process.cwd()` expression in your code, you can set this
option to tell `turbotrace` the value of `process.cwd()` while tracing.
      // for example the require(process.cwd() + '/package.json') will be traced as
require('/path/to/cwd/package.json')
      processCwd?: string
      // control the maximum memory usage of the `turbotrace`, in `MB`, default
is `6000`.
```

```
      memoryLimit?: number
    },
  },
}
```

---
title: pageExtensions
description: Extend the default page extensions used by Next.js when resolving pages in the Pages Router.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

<AppOnly>

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 filename="next.config.js"
const withMDX = require('@next/mdx')()

/** @type {import('next').NextConfig} */
const nextConfig = {
  pageExtensions: ['ts', 'tsx', 'mdx'],
  experimental: {
    mdxRs: true,
  },
}

module.exports = withMDX(nextConfig)
```

</AppOnly>

<PagesOnly>

You can extend the default Page extensions (`.tsx`, `.ts`, `.jsx`, `.js`) used by Next.js. Inside `next.config.js`, add the `pageExtensions` config:

```js filename="next.config.js"
module.exports = {
  pageExtensions: ['mdx', 'md', 'jsx', 'js', 'tsx', 'ts'],
```

}

Changing these values affects _all_ Next.js pages, including the following:

- [`middleware.js`](/docs/pages/building-your-application/routing/middleware)
- [`instrumentation.js`](/docs/pages/building-your-application/optimizing/instrumentation)
- `pages/_document.js`
- `pages/_app.js`
- `pages/api/`

For example, if you reconfigure `.ts` page extensions to `.page.ts`, you would need to rename pages like `middleware.page.ts`, `instrumentation.page.ts`, `_app.page.ts`.

## Including non-page files in the `pages` directory

You can colocate test files or other files used by components in the `pages` directory. Inside `next.config.js`, add the `pageExtensions` config:

```js filename="next.config.js"
module.exports = {
  pageExtensions: ['page.tsx', 'page.ts', 'page.jsx', 'page.js'],
}
```

Then, rename your pages to have a file extension that includes `.page` (e.g. rename `MyPage.tsx` to `MyPage.page.tsx`). Ensure you rename _all_ Next.js pages, including the files mentioned above.

</PagesOnly>
---
title: Partial Prerendering (experimental)
description: Learn how to enable Partial Prerendering (experimental) in Next.js 14.

> **Warning**: Partial Prerendering is an experimental feature and is currently **not suitable for production environments**.

Partial Prerendering is an experimental feature that allows static portions of a route to be prerendered and served from the cache with dynamic holes streamed in, all in a single HTTP request.

Partial Prerendering is available in `next@canary`:

```bash filename="Terminal"
npm install next@canary
```

You can enable Partial Prerendering by setting the experimental `ppr` flag:

```js filename="next.config.js"
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    ppr: true,
  },
}

module.exports = nextConfig
```

> **Good to know:**
>
> - Partial Prerendering does not yet apply to client-side navigations. We are actively working on this.
> - Partial Prerendering is designed for the [Node.js runtime](/docs/app/building-your-application/rendering/edge-and-nodejs-runtimes) only. Using the subset of the Node.js runtime is not needed when you can instantly serve the static shell.

Learn more about Partial Prerendering in the [Next.js Learn course](/learn/dashboard-app/partial-prerendering).

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  poweredByHeader: false,
```

}
~~~
---
title: productionBrowserSourceMaps
description: Enables browser source map generation during the production build.
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  productionBrowserSourceMaps: true,
}
~~~

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`

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

> **Good to know**: Since Next.js 13.4, Strict Mode is `true` by default with `app` router, so the above configuration is only necessary for `pages`. You can still disable Strict Mode by setting `reactStrictMode: false`.

> **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](https://react.dev/reference/react/StrictMode) 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 filename="next.config.js"
module.exports = {
  reactStrictMode: true,
}
```

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>`.

---
title: redirects
description: Add redirects to your Next.js app.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  async redirects() {
    return [
      {
        source: '/about',
        destination: '/',
        permanent: true,
      },
    ]
```

  },
}
~~~

`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](#header-cookie-and-query-matching) with the `type`, `key` and `value` properties.
- `missing` is an array of [missing objects](#header-cookie-and-query-matching) 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](/docs/app/building-your-application/routing/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:

```js
{
  source: '/old-blog/:path*',
  destination: '/blog/:path*',
  permanent: false
~~~

}

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 filename="next.config.js"
module.exports = {
  async redirects() {
    return [
      {
        source: '/old-blog/:slug',
        destination: '/news/:slug', // Matched parameters can be used in the
destination
        permanent: true,
      },
    ]
  },
}
```

### 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 filename="next.config.js"
module.exports = {
  async redirects() {
    return [
      {
        source: '/blog/:slug*',
        destination: '/news/:slug*', // Matched parameters can be used in the
destination
        permanent: true,
      },
    ]
  },
}
```

### 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 filename="next.config.js"
module.exports = {
  async redirects() {
    return [
      {
        source: '/post/:slug(||d{1,})',
        destination: '/news/:slug', // Matched parameters can be used in the
destination
        permanent: false,
      },
    ]
  },
}
```

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 filename="next.config.js"
module.exports = {
  async redirects() {
    return [
      {
        // this will match `/english(default)/something` being requested
        source: '/english||(default||)/:slug',
        destination: '/en-us/:slug',
        permanent: false,
      },
    ]
  },
}
```

## 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 filename="next.config.js"
module.exports = {
  async redirects() {
    return [
      // if the header `x-redirect-me` is present,
      // this redirect will be applied
      {
        source: '/:path((?!another-page$).*)',
        has: [
          {
            type: 'header',
            key: 'x-redirect-me',
          },
        ],
        permanent: false,
        destination: '/another-page',
      },
      // if the header `x-dont-redirect` is present,
      // this redirect will NOT be applied
      {
        source: '/:path((?!another-page$).*)',
        missing: [
          {
            type: 'header',
            key: 'x-do-not-redirect',
          },
        ],
        permanent: false,
        destination: '/another-page',
      },
      // if the source, query, and cookie are matched,
      // this redirect will be applied
      {
        source: '/specific/:path*',
        has: [
          {
            type: 'query',
            key: 'page',
            // the page value will not be available in the
            // destination since value is provided and doesn't
            // use a named capture group e.g. (?<page>home)
            value: 'home',
          },
          {
            type: 'cookie',
            key: 'authorized',
            value: 'true',
          },
        ],
        permanent: false,
        destination: '/another/:path*',
      },
      // if the header `x-authorized` is present and
      // contains a matching value, this redirect will be applied
      {
        source: '/',
        has: [
          {
            type: 'header',
            key: 'x-authorized',
            value: '(?<authorized>yes|true)',
          },
        ],
        permanent: false,
        destination: '/home?authorized=:authorized',
      },
      // if the host is `example.com`,
      // this redirect will be applied
      {
        source: '/:path((?!another-page$).*)',
        has: [
          {
            type: 'host',
            value: 'example.com',
          },
        ],
        permanent: false,
        destination: '/another-page',
      },
    ]
  },
}
```

### Redirects with basePath support

When leveraging [`basePath` support](/docs/app/api-reference/next-config-js/basePath) with redirects each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the redirect:

```js filename="next.config.js"
module.exports = {
  basePath: '/docs',

  async redirects() {
    return [
      {
        source: '/with-basePath', // automatically becomes /docs/with-basePath
        destination: '/another', // automatically becomes /docs/another
        permanent: false,
      },
      {
        // does not add /docs since basePath: false is set
        source: '/without-basePath',
        destination: 'https://example.com',
        basePath: false,
        permanent: false,
      },
    ]
  },
}
```

### Redirects with i18n support

<AppOnly>

When leveraging [`i18n` support](/docs/app/building-your-application/routing/internationalization) 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.

</AppOnly>

<PagesOnly>

When leveraging [`i18n` support](/docs/pages/building-your-application/routing/internationalization) 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.

</PagesOnly>

```js filename="next.config.js"
module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },

  async redirects() {
    return [
      {
        source: '/with-locale', // automatically handles all locales
        destination: '/another', // automatically passes the locale on
        permanent: false,
      },
      {
        // does not handle locales automatically since locale: false is set
        source: '/nl/with-locale-manual',
        destination: '/nl/another',
        locale: false,
        permanent: false,
      },
      {
        // this matches '/' since `en` is the defaultLocale
        source: '/en',
        destination: '/en/another',
        locale: false,
        permanent: false,
      },
      // it's possible to match all locales even when locale: false is set
      {
        source: '/:locale/page',
        destination: '/en/newpage',
        permanent: false,
        locale: false,
      },
      {
        // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
        // `/` or `/fr` routes like /:path* would
        source: '/(.*)',
        destination: '/another',
        permanent: false,
      },
    ]
  },
```

}
```

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 to ensure IE11 compatibility, a `Refresh` header is automatically added for the 308 status code.

Other Redirects

- Inside [API Routes](/docs/pages/api-reference/functions/next-server), you can use `res.redirect()`.
- Inside [`getStaticProps`](/docs/pages/building-your-application/data-fetching/get-static-props) and [`getServerSideProps`](/docs/pages/building-your-application/data-fetching/get-server-side-props), you can redirect specific pages at request-time.

Version History

Version	Changes
`v13.3.0`	`missing` added.
`v10.2.0`	`has` added.
`v9.5.0`	`redirects` added.

---
title: rewrites
description: Add rewrites to your Next.js app.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

Rewrites allow you to map an incoming request path to a different destination path.

<AppOnly>

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](/docs/app/api-reference/next-config-js/redirects) will reroute to a new page and show the URL changes.

</AppOnly>
```

<PagesOnly>

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](/docs/pages/api-reference/next-config-js/redirects) will reroute to a new page and show the URL changes.

</PagesOnly>

To use rewrites you can use the `rewrites` key in `next.config.js`:

```js filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/about',
        destination: '/',
      },
    ]
  },
}
```

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](#header-cookie-and-query-matching) with the `type`, `key` and `value` properties.
- `missing` is an array of [missing objects](#header-cookie-and-query-matching) 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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return {
      beforeFiles: [
        // These rewrites are checked after headers/redirects
        // and before all files including _next/public files which
        // allows overriding page files
        {
          source: '/some-page',
          destination: '/somewhere-else',
          has: [{ type: 'query', key: 'overrideMe' }],
        },
      ],
      afterFiles: [
        // These rewrites are checked after pages/public files
        // are checked but before dynamic routes
        {
          source: '/non-existent',
          destination: '/somewhere-else',
        },
      ],
      fallback: [
        // These rewrites are checked after both pages/public files
        // and dynamic routes are checked
        {
          source: '/:path*',
          destination: `https://my-old-site.com/:path*`,
        },
      ],
    }
  },
}
```

> **Good to know**: 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:

<AppOnly>

1. [headers](/docs/app/api-reference/next-config-js/headers) are checked/applied

2. [redirects](/docs/app/api-reference/next-config-js/redirects) are checked/applied
3. `beforeFiles` rewrites are checked/applied
4. static files from the [public directory](/docs/app/building-your-application/optimizing/static-assets), `_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'](/docs/pages/api-reference/functions/get-static-paths#fallback-true) in `getStaticPaths`, the fallback `rewrites` defined in your `next.config.js` will _not_ be run.

</AppOnly>

<PagesOnly>

1. [headers](/docs/pages/api-reference/next-config-js/headers) are checked/applied
2. [redirects](/docs/pages/api-reference/next-config-js/redirects) are checked/applied
3. `beforeFiles` rewrites are checked/applied
4. static files from the [public directory](/docs/pages/building-your-application/optimizing/static-assets), `_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'](/docs/pages/api-reference/functions/get-static-paths#fallback-true) in `getStaticPaths`, the fallback `rewrites` defined in your `next.config.js` will _not_ be run.

</PagesOnly>

## 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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-about/:path*',
        destination: '/about', // The :path parameter isn't used here so will be automatically passed in the query
      },
    ]
  },
}
```

If a parameter is used in the destination none of the parameters will be automatically passed in the query.

```js filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/docs/:path*',
        destination: '/:path*', // The :path parameter is used here so will not be automatically passed in the query
      },
    ]
  },
}
```

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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/:first/:second',
        destination: '/:first?second=:second',
        // Since the :first parameter is used in the destination the :second parameter
        // will not automatically be added in the query although we can manually add it
        // as shown above
      },
    ]
  },
}
```

> **Good to know**: Static pages from [Automatic Static Optimization](/docs/

pages/building-your-application/rendering/automatic-static-optimization) or [prerendering](/docs/pages/building-your-application/data-fetching/get-static-props) 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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog/:slug',
        destination: '/news/:slug', // Matched parameters can be used in the destination
      },
    ]
  },
}
```

### 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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog/:slug*',
        destination: '/news/:slug*', // Matched parameters can be used in the destination
      },
    ]
  },
}
```

### 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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/old-blog/:post(||d{1,})',
        destination: '/blog/:post', // Matched parameters can be used in the
destination
      },
    ]
  },
}
```

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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        // this will match `/english(default)/something` being requested
        source: '/english||(default||)/:slug',
        destination: '/en-us/:slug',
      },
    ]
  },
}
```

## 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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      // if the header `x-rewrite-me` is present,
      // this rewrite will be applied
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-rewrite-me',
          },
        ],
        destination: '/another-page',
      },
      // if the header `x-rewrite-me` is not present,
      // this rewrite will be applied
      {
        source: '/:path*',
        missing: [
          {
            type: 'header',
            key: 'x-rewrite-me',
          },
        ],
        destination: '/another-page',
      },
      // if the source, query, and cookie are matched,
      // this rewrite will be applied
      {
        source: '/specific/:path*',
        has: [
          {
            type: 'query',
            key: 'page',
            // the page value will not be available in the
            // destination since value is provided and doesn't
            // use a named capture group e.g. (?<page>home)
            value: 'home',
          },
          {
            type: 'cookie',
            key: 'authorized',
            value: 'true',
          },
        ],
        destination: '/:path*/home',
      },
      // if the header `x-authorized` is present and
      // contains a matching value, this rewrite will be applied
      {
        source: '/:path*',
        has: [
          {
            type: 'header',
            key: 'x-authorized',
            value: '(?<authorized>yes|true)',
          },
        ],
        destination: '/home?authorized=:authorized',
      },
      // if the host is `example.com`,
      // this rewrite will be applied
      {
        source: '/:path*',
        has: [
          {
            type: 'host',
            value: 'example.com',
          },
        ],
        destination: '/another-page',
      },
    ]
  },
}
```

## Rewriting to an external URL

<details>
  <summary>Examples</summary>

- [Incremental adoption of Next.js](https://github.com/vercel/next.js/tree/canary/examples/custom-routes-proxying)
- [Using Multiple Zones](https://github.com/vercel/next.js/tree/canary/examples/with-zones)

</details>

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 filename="next.config.js"
module.exports = {
  async rewrites() {
    return [
      {
        source: '/blog',
        destination: 'https://example.com/blog',
      },
      {
        source: '/blog/:slug',
        destination: 'https://example.com/blog/:slug', // Matched parameters can
be used in the destination
      },
    ]
  },
}
```

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 filename="next.config.js"
module.exports = {
  trailingSlash: true,
  async rewrites() {
    return [
      {
        source: '/blog/',
        destination: 'https://example.com/blog/',
      },
      {
        source: '/blog/:path*/',
        destination: 'https://example.com/blog/:path*/',
      },
    ]
  },
}
```

### 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

```js filename="next.config.js"
module.exports = {
  async rewrites() {
    return {
      fallback: [
        {
          source: '/:path*',
          destination: `https://custom-routes-proxying-
endpoint.vercel.app/:path*`,
        },
      ],
    }
  },
}
```

### Rewrites with basePath support

When leveraging [`basePath` support](/docs/app/api-reference/next-config-js/basePath) with rewrites each `source` and `destination` is automatically prefixed with the `basePath` unless you add `basePath: false` to the rewrite:

```js filename="next.config.js"
module.exports = {
  basePath: '/docs',

  async rewrites() {
    return [
      {
        source: '/with-basePath', // automatically becomes /docs/with-basePath
        destination: '/another', // automatically becomes /docs/another
      },
      {
        // does not add /docs to /without-basePath since basePath: false is set
        // Note: this can not be used for internal rewrites e.g. `destination: '/
another'`
        source: '/without-basePath',
        destination: 'https://example.com',
        basePath: false,
      },
    ]
  },
}
```

~~~

Rewrites with i18n support

<AppOnly>

When leveraging [`i18n` support](/docs/app/building-your-application/routing/internationalization) 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.

</AppOnly>

<PagesOnly>

When leveraging [`i18n` support](/docs/pages/building-your-application/routing/internationalization) 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.

</PagesOnly>

```js filename="next.config.js"
module.exports = {
  i18n: {
    locales: ['en', 'fr', 'de'],
    defaultLocale: 'en',
  },

  async rewrites() {
    return [
      {
        source: '/with-locale', // automatically handles all locales
        destination: '/another', // automatically passes the locale on
      },
      {
        // does not handle locales automatically since locale: false is set
        source: '/nl/with-locale-manual',
        destination: '/nl/another',
        locale: false,
      },
      {
        // this matches '/' since `en` is the defaultLocale
        source: '/en',
        destination: '/en/another',
~~~

```
        locale: false,
      },
      {
        // it's possible to match all locales even when locale: false is set
        source: '/:locale/api-alias/:path*',
        destination: '/api/:path*',
        locale: false,
      },
      {
        // this gets converted to /(en|fr|de)/(.*) so will not match the top-level
        // `/` or `/fr` routes like /:path* would
        source: '/(.*)',
        destination: '/another',
      },
    ]
  },
}
```

## Version History

Version	Changes
`v13.3.0`	`missing` added.
`v10.2.0`	`has` added.
`v9.5.0`	Headers added.

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

Dependencies used inside [Server Components](/docs/app/building-your-application/rendering/server-components) and [Route Handlers](/docs/app/building-your-application/routing/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 filename="next.config.js"
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    serverComponentsExternalPackages: ['@acme/ui'],
  },
}

module.exports = nextConfig
```

Next.js includes a [short list of popular packages](https://github.com/vercel/next.js/blob/canary/packages/next/src/lib/server-external-packages.json) that currently are working on compatibility and automatically opt-ed out:

- `@aws-sdk/client-s3`
- `@aws-sdk/s3-presigned-post`
- `@blockfrost/blockfrost-js`
- `@libsql/client`
- `@jpg-store/lucid-cardano`
- `@mikro-orm/core`
- `@mikro-orm/knex`
- `@prisma/client`
- `@sentry/nextjs`
- `@sentry/node`
- `@swc/core`
- `argon2`
- `autoprefixer`
- `aws-crt`
- `bcrypt`
- `better-sqlite3`
- `canvas`
- `cpu-features`
- `cypress`
- `eslint`
- `express`
- `firebase-admin`
- `jest`
- `jsdom`
- `libsql`
- `lodash`
- `mdx-bundler`
- `mongodb`
- `mongoose`
- `next-mdx-remote`
- `next-seo`
- `payload`
- `pg`
- `playwright`
- `postcss`
- `prettier`
- `prisma`

- `puppeteer`
- `rimraf`
- `sharp`
- `shiki`
- `sqlite3`
- `tailwindcss`
- `ts-node`
- `typescript`
- `vscode-oniguruma`
- `webpack`
---
title: trailingSlash
description: Configure Next.js pages to resolve with or without a trailing slash.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  trailingSlash: true,
}
```

With this option set, urls like `/about` will redirect to `/about/`.

## Version History

Version	Changes
`v9.5.0`	`trailingSlash` added.

---
title: transpilePackages
description: Automatically transpile and bundle dependencies from local packages (like monorepos) or from external dependencies (`node_modules`).

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
/** @type {import('next').NextConfig} */
const nextConfig = {
  transpilePackages: ['@acme/ui', 'lodash-es'],
}

module.exports = nextConfig
```

## Version History

Version	Changes
`v13.0.0`	`transpilePackages` added.

---
title: turbo (Experimental)
nav_title: turbo
description: Configure Next.js with Turbopack-specific options
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

> **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 filename="next.config.js"
module.exports = {
  experimental: {
    turbo: {
      rules: {
        // Option format
        '*.md': [
          {
            loader: '@mdx-js/loader',
            options: {
              format: 'md',
            },
          },
        ],
        // Option-less format
        '*.mdx': ['@mdx-js/loader'],
      },
    },
  },
}
```

Then, given the above configuration, you can use transformed code from your app:

```js
import MyDoc from './my-doc.mdx'

export default function Home() {
  return <MyDoc />
}
```

## Resolve Alias

Through `next.config.js`, Turbopack can be configured to modify module resolution through aliases, similar to webpack's [`resolve.alias`](https://webpack.js.org/configuration/resolve/#resolvealias) configuration.

To configure resolve aliases, map imported patterns to their new destination in `next.config.js`:

```js filename="next.config.js"
module.exports = {
  experimental: {
    turbo: {
      resolveAlias: {
        underscore: 'lodash',
        mocha: { browser: 'mocha/browser-entry.js' },
      },
    },
  },
}
```

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](https://nodejs.org/docs/latest-v18.x/api/packages.html#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](https://turbo.build/pack/docs/migrating-from-webpack).

---
title: typedRoutes (experimental)
nav_title: typedRoutes
description: Enable experimental support for statically typed links.

Experimental support for [statically typed links](/docs/app/building-your-application/configuring/typescript#statically-typed-links). This feature requires using the App Router as well as TypeScript in your project.

```js filename="next.config.js"
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    typedRoutes: true,
  },
}

module.exports = nextConfig
```
---
title: typescript
description: Next.js reports TypeScript errors by default. Learn to opt-out of

this behavior here.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

---
title: urlImports
description: Configure Next.js to allow importing modules from external URLs (experimental).

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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 filename="next.config.js"
module.exports = {
  experimental: {
    urlImports: ['https://example.com/assets/', 'https://cdn.skypack.dev'],
  },
}
```

Then, you can import modules directly from URLs:

```js
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](/docs/app/api-reference/edge).

## 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

```js
import confetti from 'https://cdn.skypack.dev/canvas-confetti'
import { useEffect } from 'react'

export default () => {
  useEffect(() => {
    confetti()
  })
  return <p>Hello</p>
}
```

### Static Image Imports

```js
import Image from 'next/image'
import logo from 'https://example.com/assets/logo.png'

export default () => (
  <div>
    <Image src={logo} placeholder="blur" />
  </div>
)
```

### URLs in CSS

```css
.className {
  background: url('https://example.com/assets/hero.jpg');
}
```

### Asset Imports

```js
const logo = new URL('https://example.com/assets/file.txt', import.meta.url)

console.log(logo.pathname)

// prints "/_next/static/media/file.a9727b5d.txt"
```

---

title: Custom Webpack Config
nav_title: webpack
description: Learn how to customize the webpack config used by Next.js
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

> **Good to know**: 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:

<AppOnly>

- [CSS imports](/docs/app/building-your-application/styling)
- [CSS modules](/docs/app/building-your-application/styling/css-modules)
- [Sass/SCSS imports](/docs/app/building-your-application/styling/sass)
- [Sass/SCSS modules](/docs/app/building-your-application/styling/sass)
- [preact](https://github.com/vercel/next.js/tree/canary/examples/using-preact)

</AppOnly>

<PagesOnly>

- [CSS imports](/docs/pages/building-your-application/styling)
- [CSS modules](/docs/pages/building-your-application/styling/css-modules)
- [Sass/SCSS imports](/docs/pages/building-your-application/styling/sass)
- [Sass/SCSS modules](/docs/pages/building-your-application/styling/sass)
- [preact](https://github.com/vercel/next.js/tree/canary/examples/using-preact)
- [Customizing babel configuration](/docs/pages/building-your-application/configuring/babel)

</PagesOnly>

Some commonly asked for features are available as plugins:

- [@next/mdx](https://github.com/vercel/next.js/tree/canary/packages/next-mdx)
- [@next/bundle-analyzer](https://github.com/vercel/next.js/tree/canary/packages/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 filename="next.config.js"
module.exports = {
  webpack: (
    config,
    { buildId, dev, isServer, defaultLoaders, nextRuntime, webpack }
  ) => {
    // Important: return the modified config
    return config
  },
}
```

> The `webpack` function is executed three times, twice for the server (nodejs / edge runtime) 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`:

```js
// Example config for adding a loader that depends on babel-loader
// This source was taken from the @next/mdx plugin source:
// https://github.com/vercel/next.js/tree/canary/packages/next-mdx
module.exports = {
  webpack: (config, options) => {
    config.module.rules.push({
      test: /|.mdx/,
      use: [
        options.defaultLoaders.babel,
        {
          loader: '@mdx-js/loader',
          options: pluginOptions.options,
        },
      ],
```

    })

    return config
  },
}
```

#### `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.

---
title: webVitalsAttribution
description: Learn how to use the webVitalsAttribution option to pinpoint the source of Web Vitals issues.
---

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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](https://github.com/GoogleChrome/web-vitals/blob/4ca38ae64b8d1e899028c692f94d4c56acfc996c/README.md#attribution), allows us to obtain more in-depth information like entries for [PerformanceEventTiming](https://developer.mozilla.org/docs/Web/API/PerformanceEventTiming), [PerformanceNavigationTiming](https://developer.mozilla.org/docs/Web/API/PerformanceNavigationTiming) and [PerformanceResourceTiming](https://developer.mozilla.org/docs/Web/API/PerformanceResourceTiming).

Attribution is disabled by default in Next.js but can be enabled **per metric** by specifying the following in `next.config.js`.
```

```js filename="next.config.js"
experimental: {
  webVitalsAttribution: ['CLS', 'LCP']
}
```

Valid attribution values are all `web-vitals` metrics specified in the [`NextWebVitalsMetric`](https://github.com/vercel/next.js/blob/442378d21dd56d6e769863eb8c2cb521a463a2e0/packages/next/shared/lib/utils.ts#L43) type.

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

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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](https://github.com/vercel/next.js/tree/canary/examples).

### Interactive

You can create a new project interactively by running:

```bash filename="Terminal"
npx create-next-app@latest
```

```bash filename="Terminal"
yarn create next-app
```

```bash filename="Terminal"
pnpm create next-app
```

```bash filename="Terminal"
```

bunx create-next-app

You will then be asked the following prompts:

```txt filename="Terminal"
What is your project named?  my-app
Would you like to use TypeScript?  No / Yes
Would you like to use ESLint?  No / Yes
Would you like to use Tailwind CSS?  No / Yes
Would you like to use `src/` directory?  No / Yes
Would you like to use App Router? (recommended)  No / Yes
Would you like to customize the default import alias (@/*)?  No / Yes
```

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`:

```bash filename="Terminal"
Usage: create-next-app <project-directory> [options]

Options:
  -V, --version                        output the version number
  --ts, --typescript

    Initialize as a TypeScript project. (default)

  --js, --javascript

    Initialize as a JavaScript project.

  --tailwind

    Initialize with Tailwind CSS config. (default)

  --eslint
  Initialize with ESLint config.

--app

  Initialize as an App Router project.

--src-dir

  Initialize inside a `src/` directory.

--import-alias <alias-to-configure>

  Specify import alias to use (default "@/*").

--use-npm

  Explicitly tell the CLI to bootstrap the app using npm

--use-pnpm

  Explicitly tell the CLI to bootstrap the app using pnpm

--use-yarn

  Explicitly tell the CLI to bootstrap the app using Yarn

--use-bun

  Explicitly tell the CLI to bootstrap the app using Bun

-e, --example [name]|[github-url]

  An example to bootstrap the app with. You can use an example name
  from the official Next.js repo or a public GitHub URL. The URL can use
  any branch and/or subdirectory

--example-path <path-to-example>

  In a rare case, your GitHub URL might contain a branch name with
  a slash (e.g. bug/fix-1) and the path to the example (e.g. foo/bar).
  In this case, you must specify the path to the example separately:
  --example-path foo/bar

--reset-preferences

  Explicitly tell the CLI to reset any stored preferences
  -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`) or any public GitHub repository.
- **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.

---
title: Edge Runtime
description: API Reference for the Edge Runtime.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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

## Network APIs

API	Description
[`Blob`](https://developer.mozilla.org/docs/Web/API/Blob)	Represents a blob
[`fetch`](https://developer.mozilla.org/docs/Web/API/Fetch_API)	Fetches a resource

[`FetchEvent`](https://developer.mozilla.org/docs/Web/API/FetchEvent)	Represents a fetch event
[`File`](https://developer.mozilla.org/docs/Web/API/File)	Represents a file
[`FormData`](https://developer.mozilla.org/docs/Web/API/FormData)	Represents form data
[`Headers`](https://developer.mozilla.org/docs/Web/API/Headers)	Represents HTTP headers
[`Request`](https://developer.mozilla.org/docs/Web/API/Request)	Represents an HTTP request
[`Response`](https://developer.mozilla.org/docs/Web/API/Response)	Represents an HTTP response
[`URLSearchParams`](https://developer.mozilla.org/docs/Web/API/URLSearchParams)	Represents URL search parameters
[`WebSocket`](https://developer.mozilla.org/docs/Web/API/WebSocket)	Represents a websocket connection

## Encoding APIs

API	Description
[`atob`](https://developer.mozilla.org/docs/Web/API/WindowOrWorkerGlobalScope/atob)	Decodes a base-64 encoded string
[`btoa`](https://developer.mozilla.org/docs/Web/API/WindowOrWorkerGlobalScope/btoa)	Encodes a string in base-64
[`TextDecoder`](https://developer.mozilla.org/docs/Web/API/TextDecoder)	Decodes a Uint8Array into a string
[`TextDecoderStream`](https://developer.mozilla.org/docs/Web/API/TextDecoderStream)	Chainable decoder for streams
[`TextEncoder`](https://developer.mozilla.org/docs/Web/API/TextEncoder)	Encodes a string into a Uint8Array
[`TextEncoderStream`](https://developer.mozilla.org/docs/Web/API/TextEncoderStream)	Chainable encoder for streams

## Stream APIs

API	Description
[`ReadableStream`](https://developer.mozilla.org/docs/Web/API/ReadableStream)	Represents a readable stream

| [`ReadableStreamBYOBReader`](https://developer.mozilla.org/docs/Web/API/

ReadableStreamBYOBReader)	Represents a reader of a ReadableStream
[`ReadableStreamDefaultReader`](https://developer.mozilla.org/docs/Web/API/ReadableStreamDefaultReader)	Represents a reader of a ReadableStream
[`TransformStream`](https://developer.mozilla.org/docs/Web/API/TransformStream)	Represents a transform stream
[`WritableStream`](https://developer.mozilla.org/docs/Web/API/WritableStream)	Represents a writable stream
[`WritableStreamDefaultWriter`](https://developer.mozilla.org/docs/Web/API/WritableStreamDefaultWriter)	Represents a writer of a WritableStream

## Crypto APIs

API	Description
[`crypto`](https://developer.mozilla.org/docs/Web/API/Window/crypto)	Provides access to the cryptographic functionality of the platform
[`CryptoKey`](https://developer.mozilla.org/docs/Web/API/CryptoKey)	Represents a cryptographic key
[`SubtleCrypto`](https://developer.mozilla.org/docs/Web/API/SubtleCrypto)	Provides access to common cryptographic primitives, like hashing, signing, encryption or decryption

## Web Standard APIs

API	Description
[`AbortController`](https://developer.mozilla.org/docs/Web/API/AbortController)	Allows you to abort one or more DOM requests as and when desired
[`Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array)	Represents an array of values
[`ArrayBuffer`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer)	Represents a generic, fixed-length raw

binary data buffer
|
| [`Atomics`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Atomics) | Provides atomic operations as static methods
|
| [`BigInt`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/BigInt) | Represents a whole number with arbitrary precision
|
| [`BigInt64Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/BigInt64Array) | Represents a typed array of 64-bit signed integers
|
| [`BigUint64Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/BigUint64Array) | Represents a typed array of 64-bit unsigned integers
|
| [`Boolean`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Boolean) | Represents a logical entity and can have two values: `true` and `false`
|
| [`clearInterval`](https://developer.mozilla.org/docs/Web/API/WindowOrWorkerGlobalScope/clearInterval) | Cancels a timed, repeating action which was previously established by a call to `setInterval()`
|
| [`clearTimeout`](https://developer.mozilla.org/docs/Web/API/WindowOrWorkerGlobalScope/clearTimeout) | Cancels a timed, repeating action which was previously established by a call to `setTimeout()`
|
| [`console`](https://developer.mozilla.org/docs/Web/API/Console)
| Provides access to the browser's debugging console
|
| [`DataView`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/DataView) | Represents a generic view of an `ArrayBuffer`
|
| [`Date`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Date) | Represents a single moment in time in a platform-independent format
|
| [`decodeURI`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/decodeURI) | Decodes a Uniform Resource Identifier (URI) previously created by `encodeURI` or by a similar routine
|
| [`decodeURIComponent`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/decodeURIComponent) | Decodes a Uniform

Resource Identifier (URI) component previously created by `encodeURIComponent` or by a similar routine
|
| [`DOMException`](https://developer.mozilla.org/docs/Web/API/DOMException) | Represents an error that occurs in the DOM
|
| [`encodeURI`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/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`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/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`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Error) | Represents an error when trying to execute a statement or accessing a property
|
| [`EvalError`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/EvalError) | Represents an error that occurs regarding the global function `eval()`
|
| [`Float32Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Float32Array) | Represents a typed array of 32-bit floating point numbers
|
| [`Float64Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Float64Array) | Represents a typed array of 64-bit floating point numbers
|
| [`Function`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Function) | Represents a function
|
| [`Infinity`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Infinity) | Represents the mathematical Infinity value
|
| [`Int8Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Int8Array) | Represents a typed array of 8-bit signed integers
|
| [`Int16Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Int16Array) | Represents a typed array of 16-bit signed integers
|

| [`Int32Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/Int32Array) | Represents a typed array of 32-bit
signed integers
|
| [`Intl`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/Intl) | Provides access to internationalization
and localization functionality
|
| [`isFinite`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/isFinite) | Determines whether a value is a finite
number
|
| [`isNaN`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/isNaN) | Determines whether a value is `NaN` or
not
|
| [`JSON`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/JSON) | Provides functionality to convert
JavaScript values to and from the JSON format
|
| [`Map`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/Map) | Represents a collection of values,
where each value may occur only once
|
| [`Math`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/Math) | Provides access to mathematical
functions and constants
|
| [`Number`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/Number) | Represents a numeric value
|
| [`Object`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/Object) | Represents the object that is the base
of all JavaScript objects
|
| [`parseFloat`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/parseFloat) | Parses a string argument and returns a
floating point number
|
| [`parseInt`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/parseInt) | Parses a string argument and returns an
integer of the specified radix
|
| [`Promise`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/
Global_Objects/Promise) | Represents the eventual completion (or
failure) of an asynchronous operation, and its resulting value
|

| [`Proxy`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Proxy) | Represents an object that is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc) |
| [`queueMicrotask`](https://developer.mozilla.org/docs/Web/API/queueMicrotask) | Queues a microtask to be executed
|
| [`RangeError`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/RangeError) | Represents an error when a value is not in the set or range of allowed values
|
| [`ReferenceError`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError) | Represents an error when a non-existent variable is referenced
|
| [`Reflect`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Reflect) | Provides methods for interceptable JavaScript operations
|
| [`RegExp`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/RegExp) | Represents a regular expression, allowing you to match combinations of characters
|
| [`Set`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Set) | Represents a collection of values, where each value may occur only once
|
| [`setInterval`](https://developer.mozilla.org/docs/Web/API/setInterval)
| Repeatedly calls a function, with a fixed time delay between each call
|
| [`setTimeout`](https://developer.mozilla.org/docs/Web/API/setTimeout)
| Calls a function or evaluates an expression after a specified number of milliseconds
|
| [`SharedArrayBuffer`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer) | Represents a generic, fixed-length raw binary data buffer
|
| [`String`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String) | Represents a sequence of characters
|
| [`structuredClone`](https://developer.mozilla.org/docs/Web/API/Web_Workers_API/Structured_clone_algorithm) | Creates a deep copy of a value
|
| [`Symbol`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Symbol) | Represents a unique and immutable

data type that is used as the key of an object property
|
| [`SyntaxError`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) | Represents an error when trying to interpret syntactically invalid code
|
| [`TypeError`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/TypeError) | Represents an error when a value is not of the expected type
|
| [`Uint8Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Uint8Array) | Represents a typed array of 8-bit unsigned integers
|
| [`Uint8ClampedArray`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Uint8ClampedArray) | Represents a typed array of 8-bit unsigned integers clamped to 0-255
|
| [`Uint32Array`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Uint32Array) | Represents a typed array of 32-bit unsigned integers
|
| [`URIError`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/URIError) | Represents an error when a global URI handling function was used in a wrong way
|
| [`URL`](https://developer.mozilla.org/docs/Web/API/URL)
| Represents an object providing static methods used for creating object URLs
|
| [`URLPattern`](https://developer.mozilla.org/docs/Web/API/URLPattern)
| Represents a URL pattern
|
| [`URLSearchParams`](https://developer.mozilla.org/docs/Web/API/URLSearchParams) | Represents a collection of key/value pairs
|
| [`WeakMap`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/WeakMap) | Represents a collection of key/value pairs in which the keys are weakly referenced
|
| [`WeakSet`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/WeakSet) | Represents a collection of objects in which each object may occur only once
|
| [`WebAssembly`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly) | Provides access to WebAssembly

|

## Next.js Specific Polyfills

- [`AsyncLocalStorage`](https://nodejs.org/api/async_context.html#class-asynclocalstorage)

## Environment Variables

You can use `process.env` to access [Environment Variables](/docs/app/building-your-application/configuring/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
[`eval`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/eval)	Evaluates JavaScript code represented as a string
[`new Function(evalString)`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Function)	Creates a new function with the code provided as an argument
[`WebAssembly.compile`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/compile)	Compiles a WebAssembly module from a buffer source
[`WebAssembly.instantiate`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/WebAssembly/instantiate)	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:

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

`unstable_allowDynamic` is a [glob](https://github.com/micromatch/micromatch#matching-features), 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_.

---
title: Next.js CLI
description: The Next.js CLI allows you to start, build, and export your application. Learn more about it here.

{/* The content of this doc is shared between the app and pages router. You can use the `<PagesOnly>Content</PagesOnly>` component to add content that is specific to the Pages Router. Any shared content should not be wrapped in a component. */}

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:

```bash filename="Terminal"
npx next -h
```

([npx](https://medium.com/@maybekatz/introducing-npx-an-npm-package-runner-55f7d4bd282b) comes with npm 5.2+ and higher)

The output should look like this:

```bash filename="Terminal"
Usage
  $ next <command>

Available commands
  build, start, export, dev, lint, telemetry, info

Options
  --version, -v   Version number
  --help, -h      Displays this message

For more information run a command with the --help flag
  $ next build --help
```

You can pass any [node arguments](https://nodejs.org/api/cli.html#cli_node_options_options) to `next` commands:

```bash filename="Terminal"
NODE_OPTIONS='--throw-deprecation' next
NODE_OPTIONS='-r esm' next
NODE_OPTIONS='--inspect' next
```

> **Good to know**: 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](https://nextjs.org/blog/next-9-5):

```bash filename="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:

```bash filename="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:

```bash filename="Terminal"
npx next dev -p 4000
```

Or using the `PORT` environment variable:

```bash filename="Terminal"
PORT=4000 npx next dev
```

> **Good to know**: `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:

```bash filename="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`](#build) first.

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

```bash filename="Terminal"
npx next start -p 4000
```

Or using the `PORT` environment variable:

```bash filename="Terminal"
PORT=4000 npx next start
```

> **Good to know**:
>
> -`PORT` cannot be set in `.env` as booting up the HTTP server happens before any other code is initialized.
>
> - `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](https://nodejs.org/api/http.html#http_server_keepalivetimeout) 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:

```bash filename="Terminal"
npx next start --keepAliveTimeout 70000
```

## Lint

`next lint` runs ESLint for all files in the `pages/`, `app/`, `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:

```bash filename="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](https://nextjs.org/telemetry/).

## 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:

```bash filename="Terminal"
next info
```

will give you information like this example:

```bash filename="Terminal"

Operating System:
  Platform: linux
  Arch: x64
  Version: #22-Ubuntu SMP Fri Nov 5 13:21:36 UTC 2021
Binaries:
  Node: 16.13.0
  npm: 8.1.0
  Yarn: 1.22.17
  pnpm: 6.24.2
Relevant packages:
  next: 12.0.8
  react: 17.0.2
  react-dom: 17.0.2
```

This information should then be pasted into GitHub Issues.

In order to diagnose installation issues, you can run `next info --verbose` to print additional information about system and the installation of next-related packages.

```
---
title: API Reference
description: Next.js API Reference for the App Router.
---
```

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