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 or later.
- macOS, Windows (including WSL), and Linux are supported.

Automatic Installation

We recommend starting a new Next.js app using `create-next-app`, which sets up everything automatically for you. To create a project, run:

```
npx create-next-app@latest
```

On installation, you'll see the following prompts:

```
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, ESLint, and Tailwind CSS configuration by default.
> - You can optionally use a `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:

```
npm install next@latest react@latest react-dom@latest
```

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

```
{
  "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` to start Next.js in development mode.
- `build`: runs `next build` to build the application for production usage.
- `start`: runs `next start` to start a Next.js production server.
- `lint`: runs `next lint` to set up Next.js' built-in ESLint configuration.

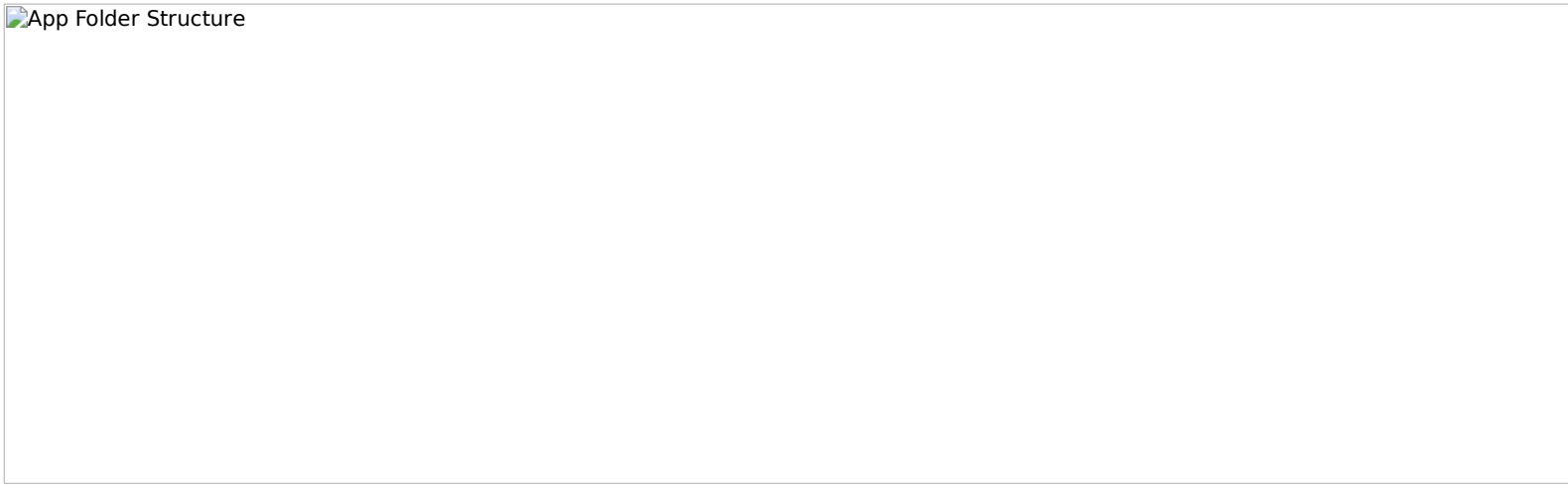
Creating directories

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. This router allows you to use React's latest features and is an evolution of the Pages Router 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 (`/`).



Create a root layout inside `app/layout.tsx` with the required `<html>` and `<body>` tags:

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

```
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:

```
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```

```
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.

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 (`/`):

```
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.

```
import type { AppProps } from 'next/app'

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

```
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.

```
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.

> **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	App Router
pages	Pages Router
public	Static assets to be served
src	Optional application source folder

Top-level files

Next.js

next.config.js	Configuration file for Next.js
package.json	Project dependencies and scripts
instrumentation.ts	OpenTelemetry and Instrumentation file
middleware.ts	Next.js request middleware
.env	Environment variables
.env.local	Local environment variables
.env.production	Production environment variables
.env.development	Development environment variables
.eslintrc.json	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	.js .jsx .tsx	Layout
page	.js .jsx .tsx	Page
loading	.js .jsx .tsx	Loading UI
not-found	.js .jsx .tsx	Not found UI
error	.js .jsx .tsx	Error UI
global-error	.js .jsx .tsx	Global error UI
route	.js .ts	API endpoint
template	.js .jsx .tsx	Re-rendered layout
default	.js .jsx .tsx	Parallel route fallback page

Nested Routes

folder	Route segment
folder/folder	Nested route segment

Dynamic Routes

[folder]	Dynamic route segment
[...folder]	Catch-all route segment
[[...folder]]	Optional catch-all route segment

Route Groups and Private Folders

(folder)	Group routes without affecting routing
_folder	Opt folder and all child segments out of routing

Parallel and Intercepted Routes

@folder	Named slot
(.)folder	Intercept same level
(..)folder	Intercept one level above
(..)(..)folder	Intercept two levels above
(...)folder	Intercept from root

Metadata File Conventions

App Icons

favicon	.ico	Favicon file
icon	.ico .jpg .jpeg .png .svg	App Icon file
icon	.js .ts .tsx	Generated App Icon
apple-icon	.jpg .jpeg, .png	Apple App Icon file
apple-icon	.js .ts .tsx	Generated Apple App Icon

Open Graph and Twitter Images

opengraph-image	.jpg .jpeg .png .gif	Open Graph image file
opengraph-image	.js .ts .tsx	Generated Open Graph image

twitter-image	.jpg .jpeg .png .gif	Twitter image file
twitter-image	.js .ts .tsx	Generated Twitter image

SEO

sitemap	.xml	Sitemap file
sitemap	.js .ts	Generated Sitemap
robots	.txt	Robots file
robots	.js .ts	Generated Robots file

pages Routing Conventions

Special Files

_app	.js .jsx .tsx	Custom App
_document	.js .jsx .tsx	Custom Document
_error	.js .jsx .tsx	Custom Error Page
404	.js .jsx .tsx	404 Error Page
500	.js .jsx .tsx	500 Error Page

Routes

Folder convention		
index	.js .jsx .tsx	Home page
folder/index	.js .jsx .tsx	Nested page
File convention		
index	.js .jsx .tsx	Home page
file	.js .jsx .tsx	Nested page

Dynamic Routes

Folder convention		
[folder]/index	.js .jsx .tsx	Dynamic route segment
[...folder]/index	.js .jsx .tsx	Catch-all route segment
[[...folder]]/index	.js .jsx .tsx	Optional catch-all route segment
File convention		
[file]	.js .jsx .tsx	Dynamic route segment
[...file]	.js .jsx .tsx	Catch-all route segment
[[...file]]	.js .jsx .tsx	Optional catch-all route segment

---

title: Getting Started description: Learn how to create full-stack web applications with Next.js.

---

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

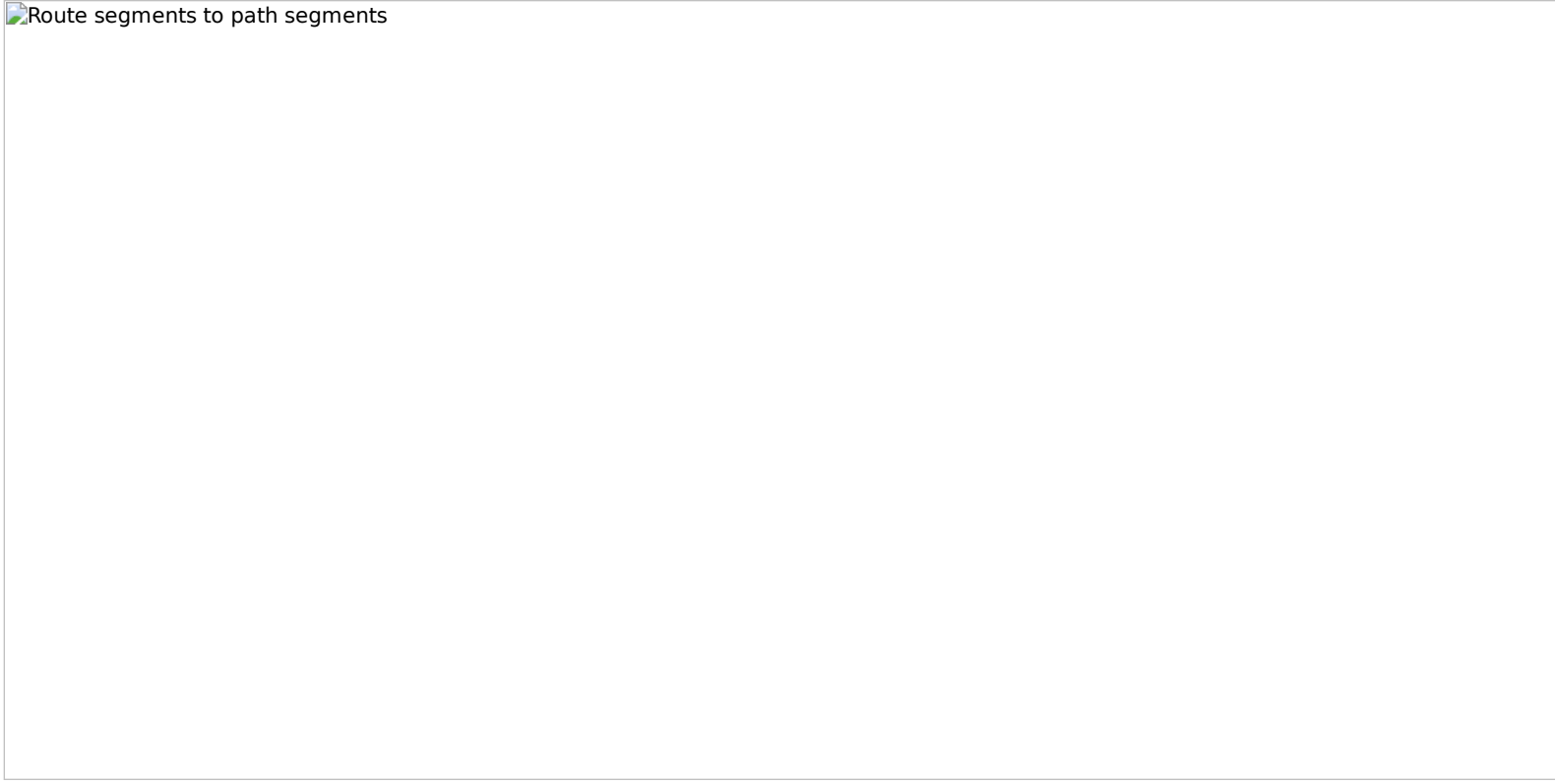
> We recommend reading the Routing Fundamentals page before continuing.

This page will guide you through how to define and organize routes in your Next.js application.

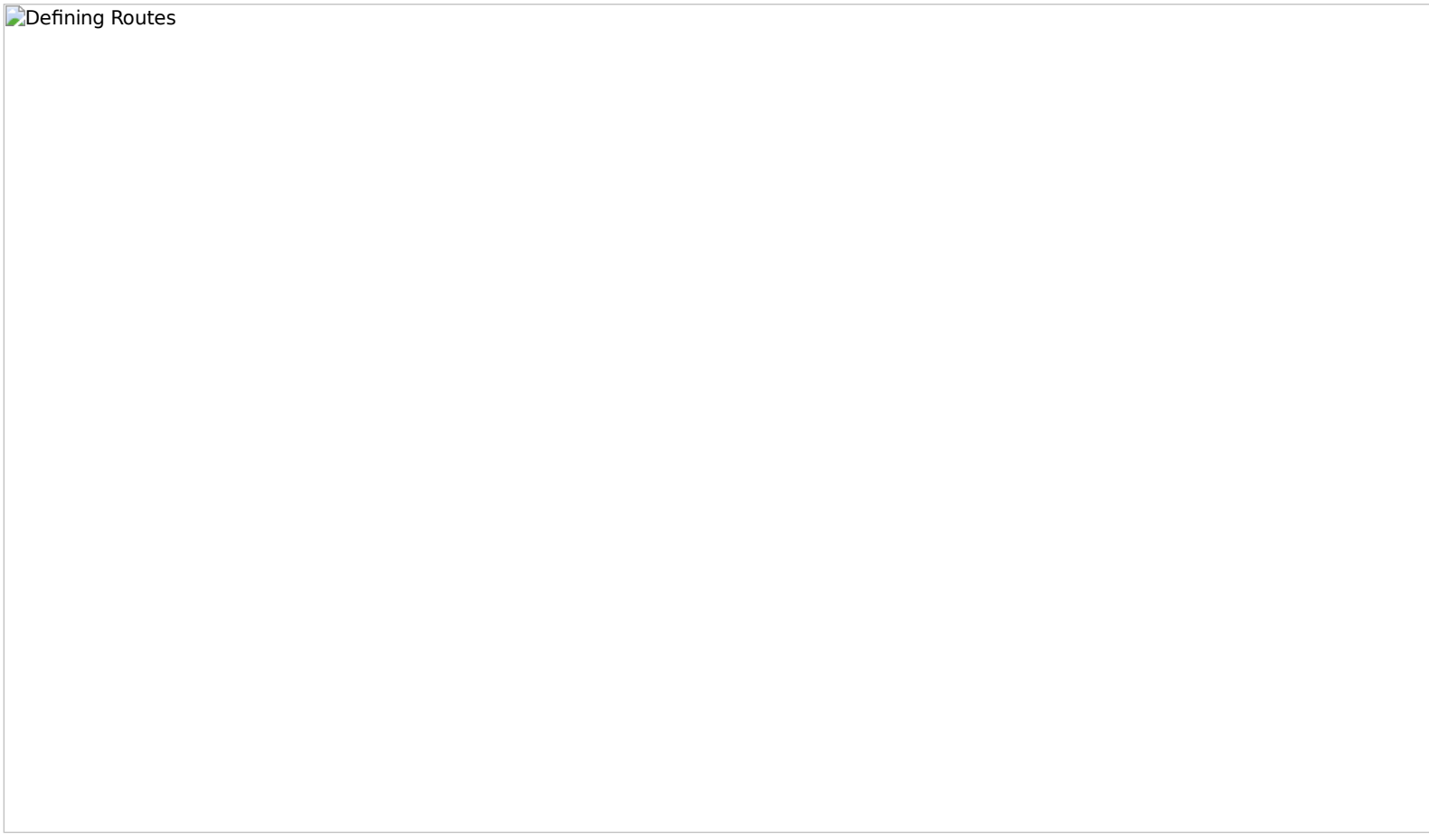
Creating Routes

Next.js uses a file-system based router where **folders** are used to define routes.

Each folder represents a **route** segment that maps to a **URL** segment. To create a nested route, you can nest folders inside each other.



A special `page.js` file is used to make route segments publicly accessible.



In this example, the `/dashboard/analytics` URL path is *not* publicly accessible because it does not have a corresponding `page.js` file. This folder could be used to store components, stylesheets, images, or other colocated files.

> **Good to know**: `.js`, `.jsx`, or `.tsx` file extensions can be used for special files.

Creating UI

Special file conventions are used to create UI for each route segment. The most common are pages to show UI unique to a route, and layouts to show UI that is shared across multiple routes.

For example, to create your first page, add a `page.js` file inside the `app` directory and export a React component:

```
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```


```
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.

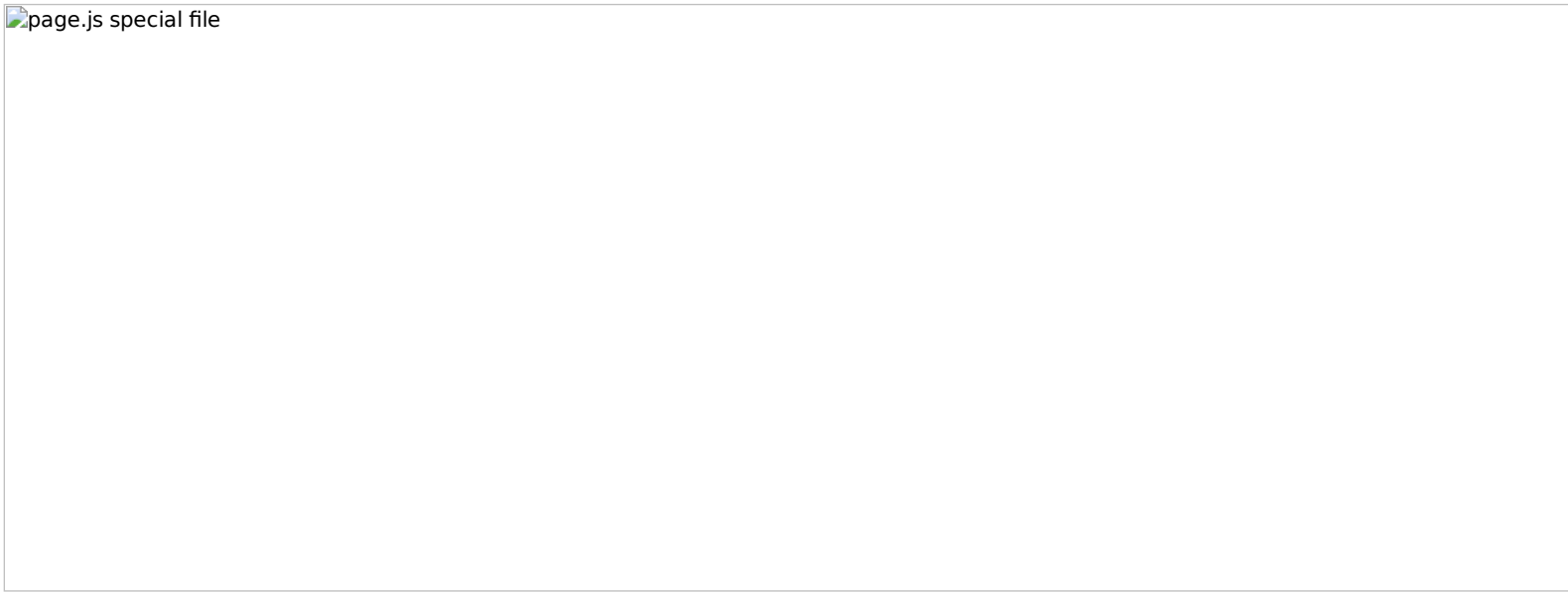
> We recommend reading the Routing Fundamentals and Defining Routes pages before continuing.

The App Router inside Next.js 13 introduced new file conventions to easily create pages, shared layouts, and templates. This page will guide you through how to use these special files in your Next.js application.

Pages

A page is UI that is **unique** to a route. You can define pages by exporting a component from a `page.js` file. Use nested folders to define a route and a `page.js` file to make the route publicly accessible.

Create your first page by adding a `page.js` file inside the `app` directory:




```
// `app/page.tsx` is the UI for the `/` URL
export default function Page() {
  return <h1>Hello, Home page!</h1>
}
```

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

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

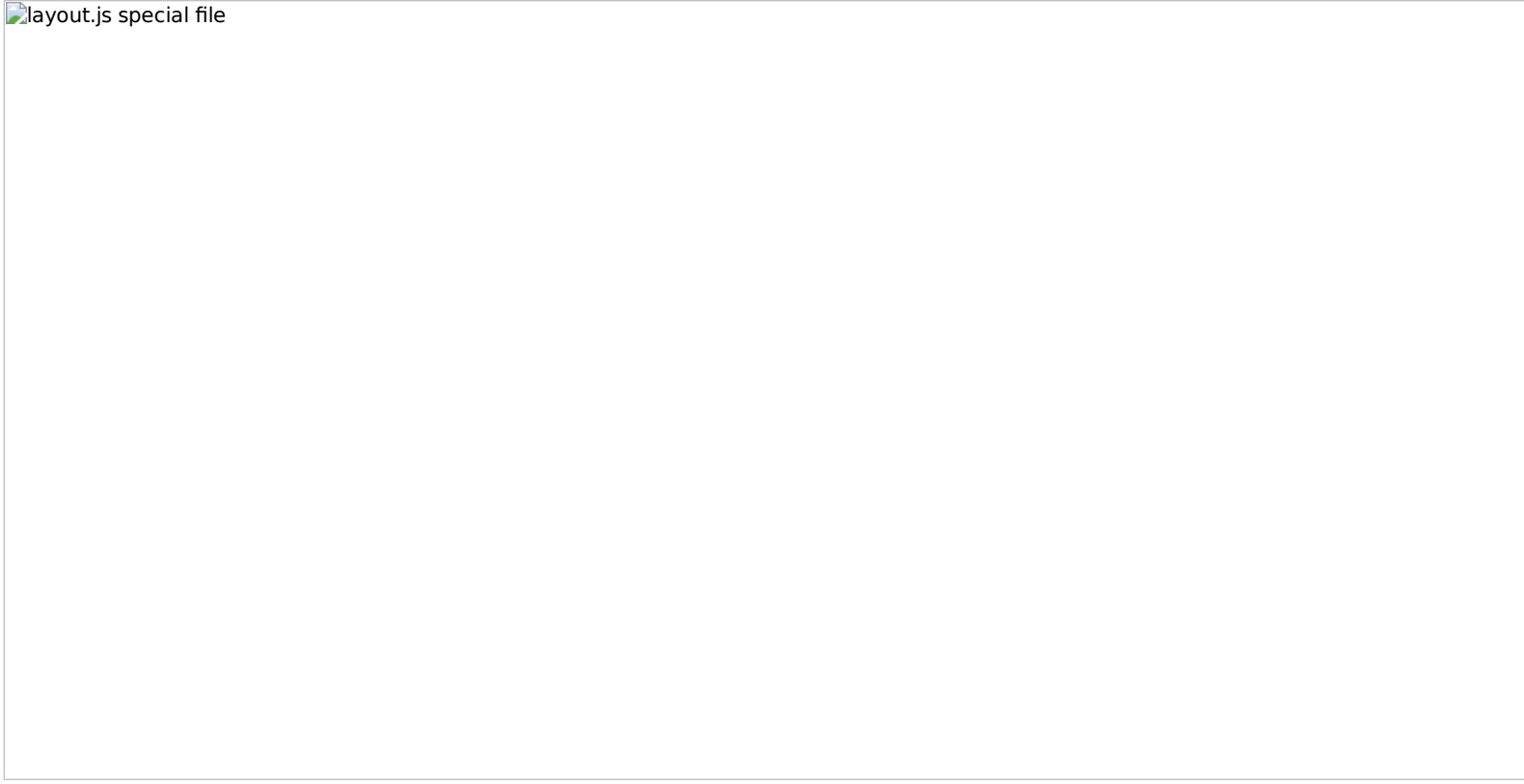
```
// `app/dashboard/page.js` is the UI for the `/dashboard` URL
export default function Page() {
  return <h1>Hello, Dashboard Page!</h1>
}
```

> **Good to know**:
>
> - A page is always the leaf of the route subtree.
> - `.js`, `.jsx`, or `.tsx` file extensions can be used for Pages.
> - A `page.js` file is required to make a route segment publicly accessible.
> - Pages are Server Components by default but can be set to a Client Component.
> - Pages can fetch data. View the Data Fetching section for more information.

Layouts

A layout is UI that is **shared** between multiple pages. On navigation, layouts preserve state, remain interactive, and do not re-render. Layouts can also be nested.

You can define a layout by `default` exporting a React component from a `layout.js` file. The component should accept a `children` prop that will be populated with a child layout (if it exists) or a child page during rendering.




```
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>
  )
}
```

```
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. This **required** layout is shared across all pages in an application. Root layouts must contain `html` and `body` tags.
- Any route segment can optionally define its own Layout. These layouts will be shared across all pages in that segment.
- Layouts in a route are **nested** by default. Each parent layout wraps child layouts below it using the React `children` prop.
- You can use Route Groups to opt specific route segments in and out of shared layouts.
- Layouts are Server Components by default but can be set to a Client Component.
- Layouts can fetch data. View the Data Fetching section for more information.
- Passing data between a parent layout and its children is not possible. However, you can fetch the same data in a route more than once, and React will automatically dedupe the requests without affecting performance.
- Layouts do not have access to the route segments below itself. To access all route segments, you can use `useSelectedLayoutSegment` or `useSelectedLayoutSegments` in a Client Component.
- `.js`, `.jsx`, or `.tsx` file extensions can be used for Layouts.
- A `layout.js` and `page.js` file can be defined in the same folder. The layout will wrap the page.

Root Layout (Required)

The root layout is defined at the top level of the `app` directory and applies to all routes. This layout enables you to modify the initial HTML returned from the server.

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

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

```
    </html>
  )
}
```

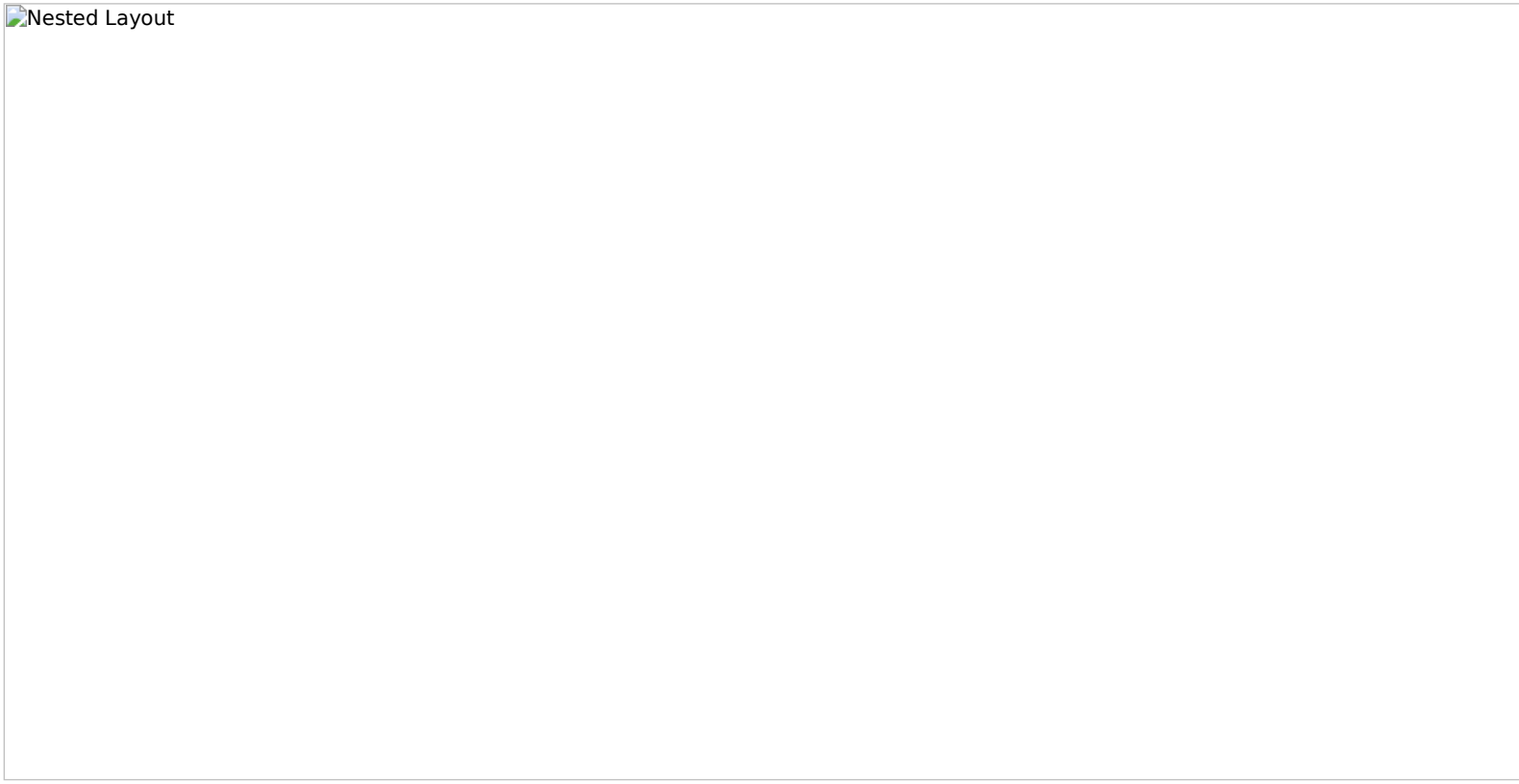
Good to know:

- 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 to manage `<head>` HTML elements, for example, the `<title>` element.
- You can use route groups to create multiple root layouts. See an example here.
- The root layout is a Server Component by default and **can not** be set to a Client Component.

Migrating from the `pages` directory: The root layout replaces the `_app.js` and `_document.js` files. View the migration guide.

Nesting Layouts

Layouts defined inside a folder (e.g. `app/dashboard/layout.js`) apply to specific route segments (e.g. `acme.com/dashboard`) and render when those segments are active. By default, layouts in the file hierarchy are **nested**, which means they wrap child layouts via their `children` prop.




```
export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return <section>{children}</section>
}
```

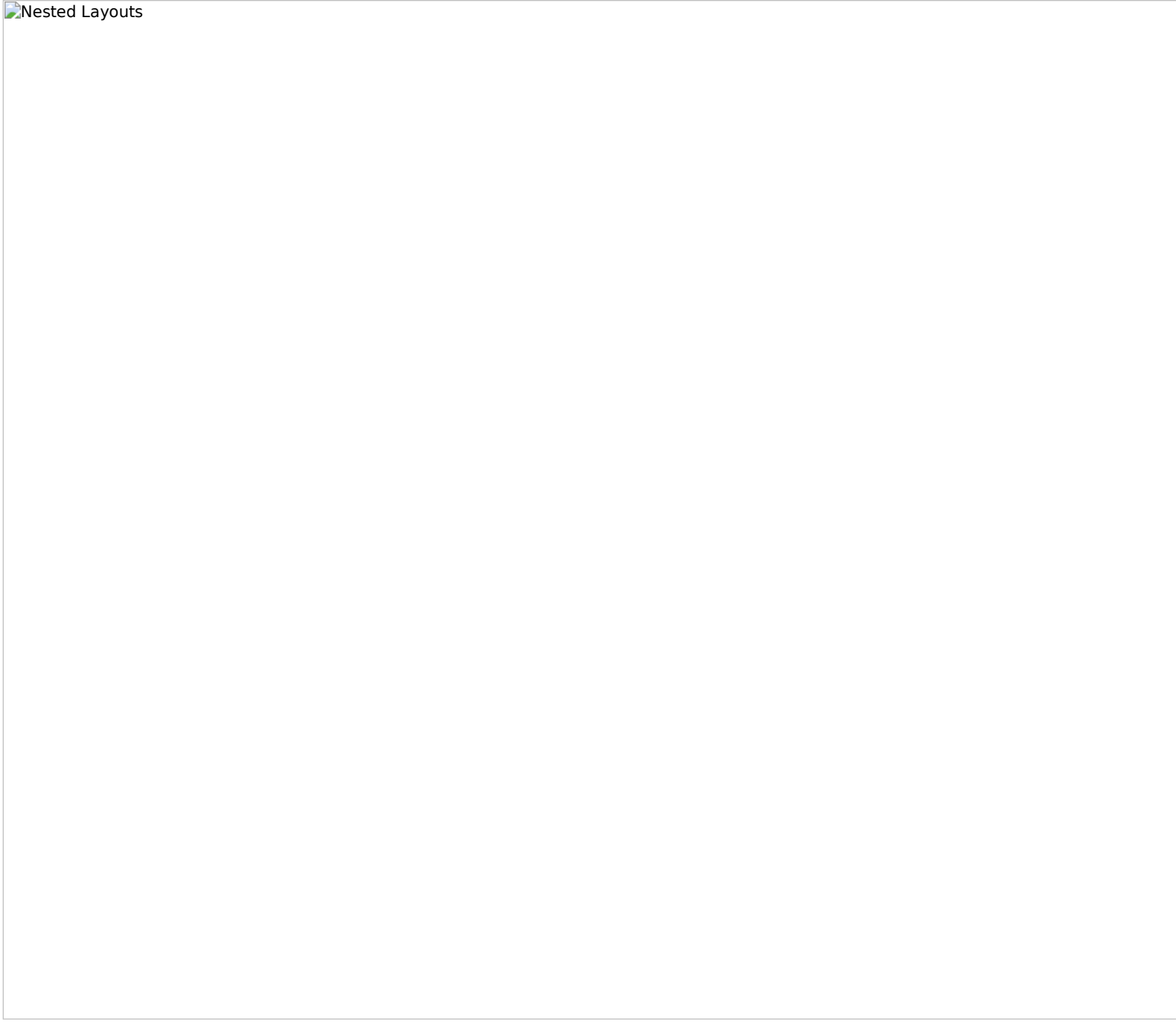
```
export default function DashboardLayout({ children }) {
  return <section>{children}</section>
}
```

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:



You can use Route Groups to opt specific route segments in and out of shared layouts.

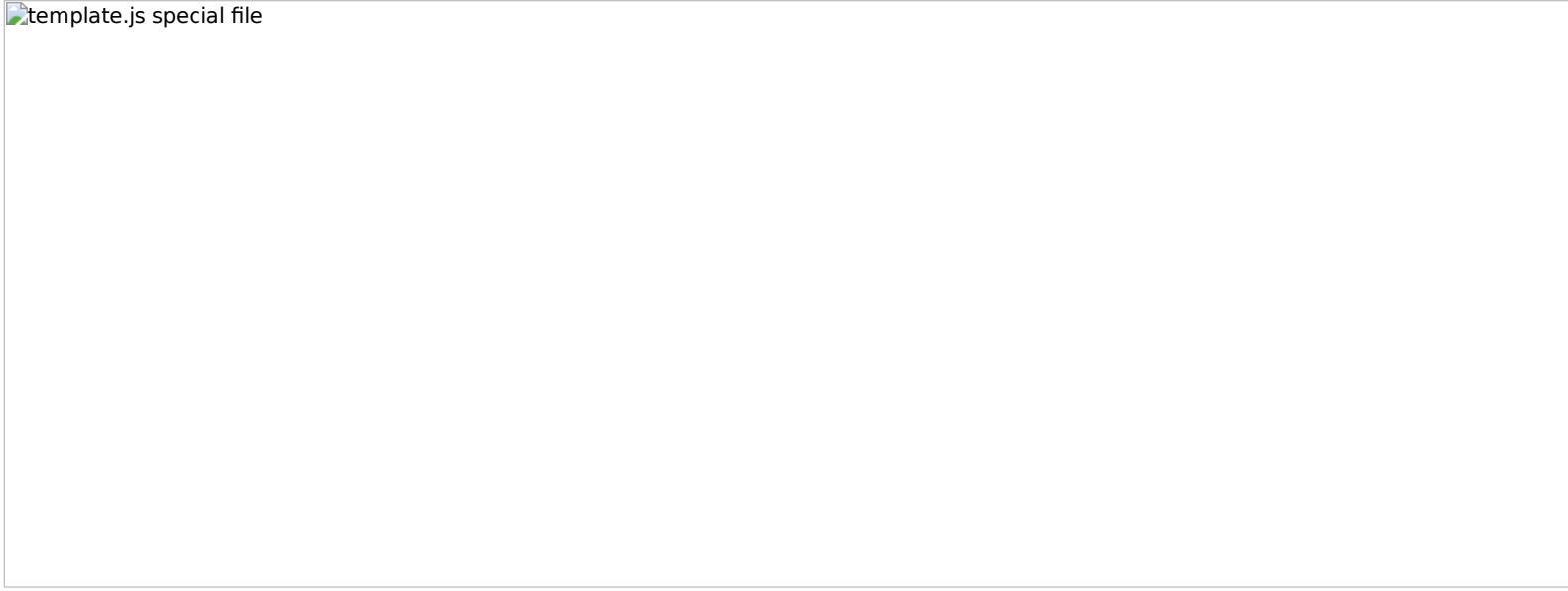
Templates

Templates are similar to layouts in that they wrap each child layout or page. Unlike layouts that persist across routes and maintain state, templates create a new instance for each of their children on navigation. This means that when a user navigates between routes that share a template, a new instance of the component is mounted, DOM elements are recreated, state is **not** preserved, and effects are re-synchronized.

There may be cases where you need those specific behaviors, and templates would be a more suitable option than layouts. For example:

- 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.




```
export default function Template({ children }: { children: React.ReactNode }) {
  return <div>{children}</div>
}

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:

```
<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.

Metadata can be defined by exporting a `metadata` object or `generateMetadata` function in a `layout.js` or `page.js` file.

```
import { Metadata } from 'next'

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

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

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 which automatically handles advanced requirements such as streaming and de-duplicating `<head>` elements.

Learn more about available metadata options in the API reference.

---

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
- Using the `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 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:

```
import Link from 'next/link'

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

```
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 for more.

Examples

Linking to Dynamic Segments

When linking to dynamic segments, you can use template literals and interpolation to generate a list of links. For example, to generate a list of blog posts:

```
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()` 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:

```
'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>
  )
}
```

```
'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.

```
<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()`.

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

// 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 from Client Components. For Server Components, you would `redirect()` instead.

```
'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.

> **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 and caches 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. Code Splitting

Code splitting allows you to split your application code into smaller bundles to be downloaded and executed by the browser. This reduces the amount of data transferred and execution time for each request, leading to improved performance.

Server Components allow your application code to be automatically code-split by route segments. This means only the code needed for the current route is loaded on navigation.

2. 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**: `prefetch` defaults to `true`. The entire route is prefetched and cached.
- **Dynamic Routes**: `prefetch` default to automatic. Only the shared layout, down the rendered "tree" of components 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 for better visual feedback to users.

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

See the `<Link>` API reference for more information.

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

3. Caching

Next.js has an **in-memory client-side cache** called the Router Cache. As users navigate around the app, the React Server Component Payload of prefetched route segments and visited routes are stored in the cache.

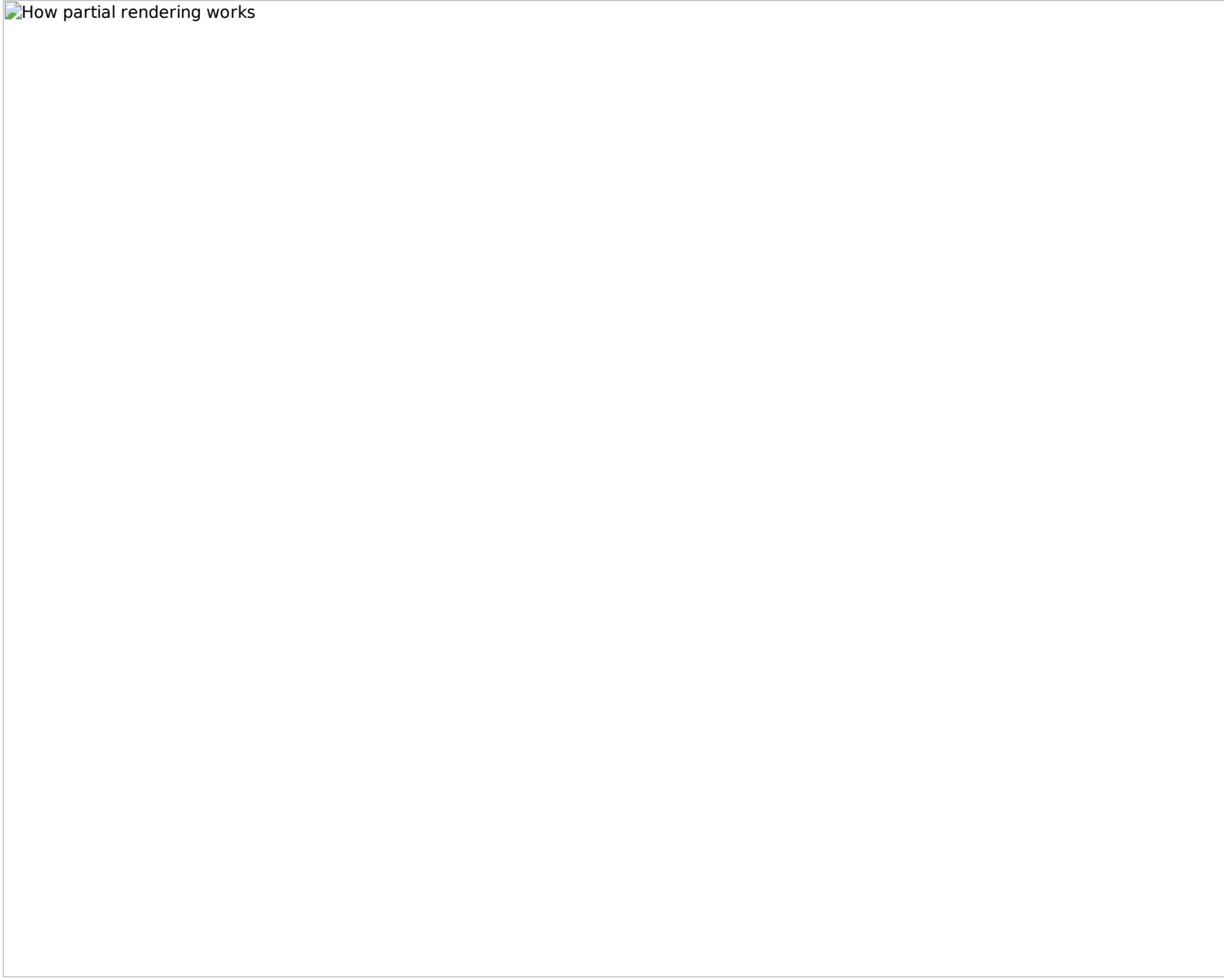
This means on navigation, the cache is reused as much as possible, instead of making a new request to the server - improving performance by reducing the number of requests and data transferred.

Learn more about how the Router Cache works and how to configure it.

4. 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.



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

5. Soft Navigation

Browsers perform a "hard navigation" when navigating between pages. The Next.js App Router enables "soft navigation" between pages, ensuring only the route segments that have changed are re-rendered (partial rendering). This enables client React state to be preserved during navigation.

6. 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.

---

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 e.g. by site section, intent, or team.
- Enabling nested layouts in the same route segment level:
  - Creating multiple nested layouts in the same segment, including multiple root layouts
  - Adding a layout to a subset of routes in a common segment

Convention

A route group can be created by wrapping a folder's name in parenthesis: `(folderName)`

Examples

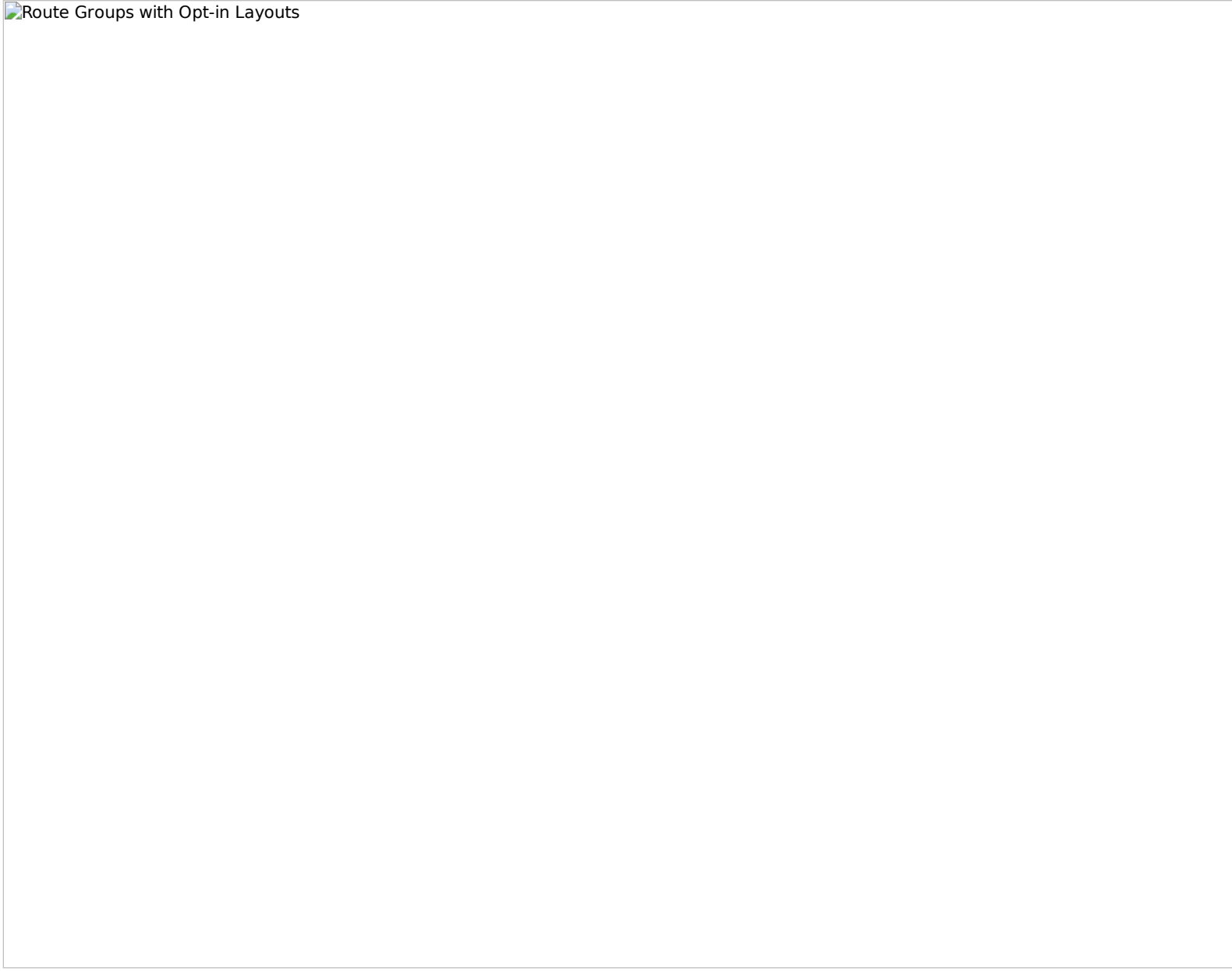
Organize routes without affecting the URL path

To organize routes without affecting the URL, create a group to keep related routes together. The folders in parenthesis will be omitted from the URL (e.g. `(marketing)` or `(shop)`).

Even though routes inside `(marketing)` and `(shop)` share the same URL hierarchy, you can create a different layout for each group by adding a `layout.js` file inside their folders.

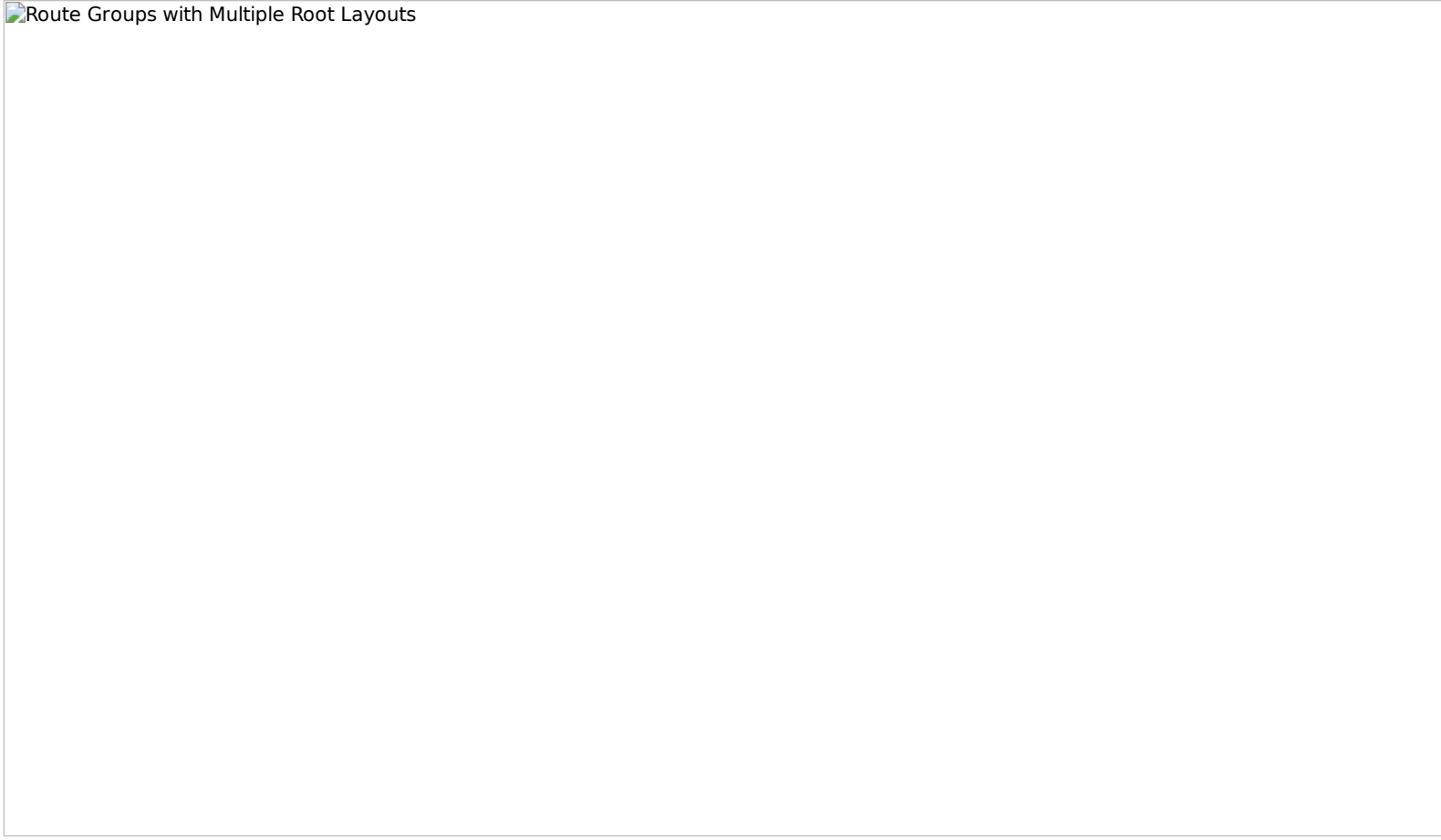
Opting specific segments into a layout

To opt specific routes into a layout, create a new route group (e.g. `(shop)`) and move the routes that share the same layout into the group (e.g. `account` and `cart`). The routes outside of the group will not share the layout (e.g. `checkout`).



Creating multiple root layouts

To create multiple root layouts, remove the top-level `layout.js` file, and add a `layout.js` file inside each route groups. This is useful for partitioning an application into sections that have a completely different UI or experience. The `<html>` and `<body>` tags need to be added to each root layout.



In the example above, both `(marketing)` and `(shop)` have their own root layout.

Good to know:

- The naming of route groups has no special significance other than for organization. They do not affect the URL path.
- Routes that include a route group **should not** resolve to the same URL path as other routes. For example, since route groups don't affect URL structure, `(marketing)/about/page.js` and `(shop)/about/page.js` would both resolve to `/about` and cause an error.
- If you use multiple root layouts without a top-level `layout.js` file, your home `page.js` file should be defined in one of the route groups, For example: `app/(marketing)/page.js`.
- Navigating **across multiple root layouts** will cause a **full page load** (as opposed to a client-side navigation). For example, navigating from `/cart` that uses `app/(shop)/layout.js` to `/blog` that uses `app/(marketing)/layout.js` will cause a full page load. This **only** applies to multiple root layouts.

---

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 at build time.

Convention

A Dynamic Segment can be created by wrapping a folder's name in square brackets: `[folderName]`. For example, `[id]` or `[slug]`.

Dynamic Segments are passed as the `params` prop to `layout`, `page`, `route`, and `generateMetadata` functions.

Example

For example, a blog could include the following route `app/blog/[slug]/page.js` where `[slug]` is the Dynamic Segment for blog posts.

```
export default function Page({ params }: { params: { slug: string } }) {
  return <div>My Post: {params.slug}</div>
}
```

```
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() page to learn how to generate the params for the segment.

Good to know: Dynamic Segments are equivalent to Dynamic Routes in the `pages` directory.

Generating Static Params

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

```
export async function generateStaticParams() {
  const posts = await fetch('https://.../posts').then((res) => res.json())

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

```
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. This means a `fetch` request with the same arguments across multiple `generateStaticParams`, Layouts, and Pages will only be made once, which decreases build times.

Use the migration guide if you are migrating from the `pages` directory.

See `generateStaticParams` server function documentation for more information and advanced use cases.

Catch-all Segments

Dynamic Segments can be extended to **catch-all** subsequent segments by adding an ellipsis inside the brackets `[...folderName]`.

For example, `app/shop/[...slug]/page.js` will match `/shop/clothes`, but also `/shop/clothes/tops`, `/shop/clothes/tops/t-shirts`, and so on.

Route	Example URL	`params`
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.

```
export default function Page({ params }: { params: { slug: string } }) {
  return <h1>My Page</h1>
}
```

```
export default function Page({ params }) {
  return <h1>My Page</h1>
}
```

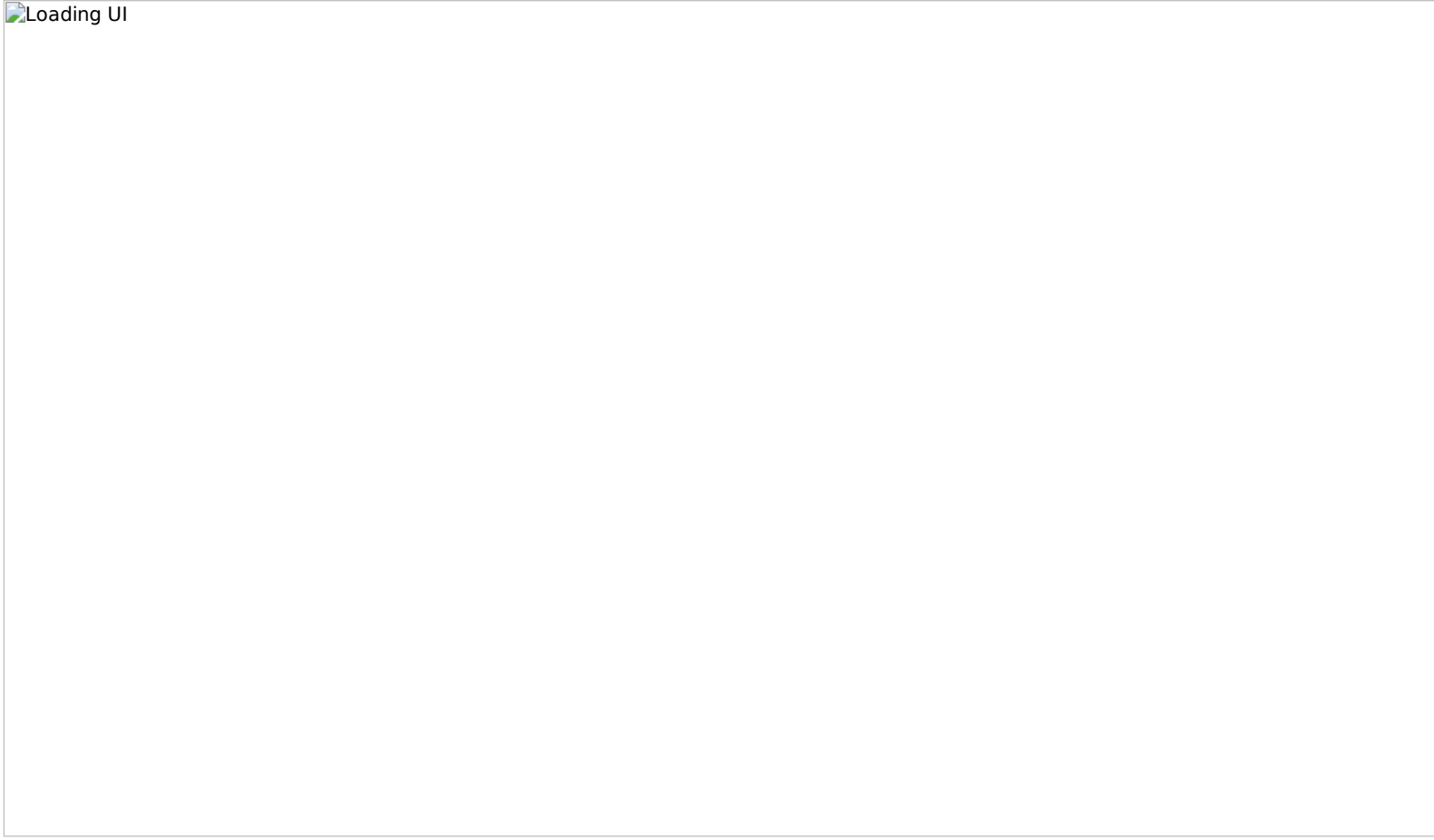
Route	`params` Type Definition
app/blog/[slug]/page.js	{ slug: string }
app/shop/[...slug]/page.js	{ slug: string[] }
app/shop/[[...slug]]/page.js	{ slug?: string[] }
app/[categoryId]/[itemId]/page.js	{ categoryId: string, itemId: string }

> **Good to know**: This may be done automatically by the TypeScript plugin 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.

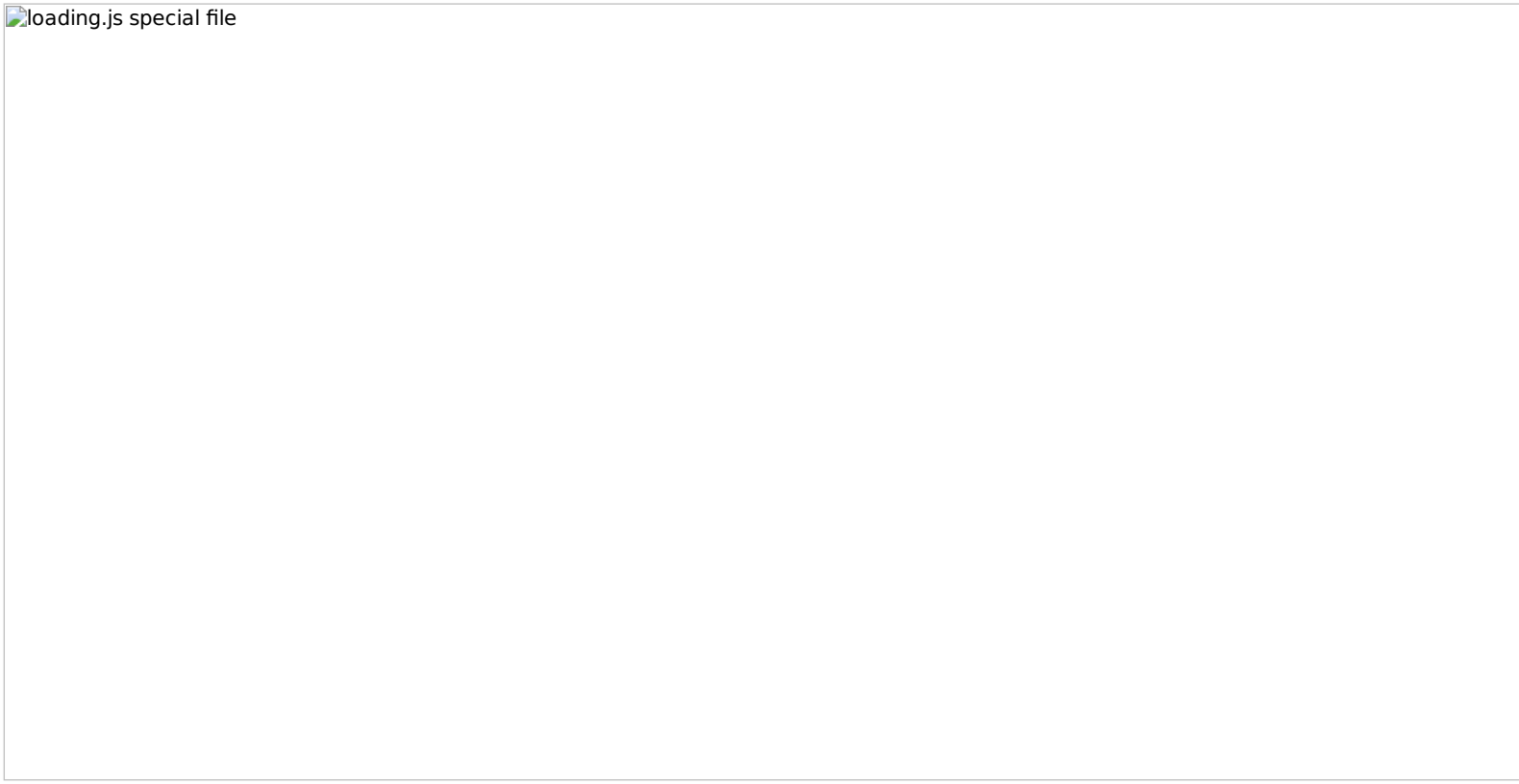
The special file `loading.js` helps you create meaningful Loading UI with React Suspense. With this convention, you can show an instant loading state from the server while the content of a route segment loads. The new content is automatically swapped in once rendering is complete.



Instant Loading States

An instant loading state is fallback UI that is shown immediately upon navigation. You can pre-render loading indicators such as skeletons and spinners, or a small but meaningful part of future screens such as a cover photo, title, etc. This helps users understand the app is responding and provides a better user experience.

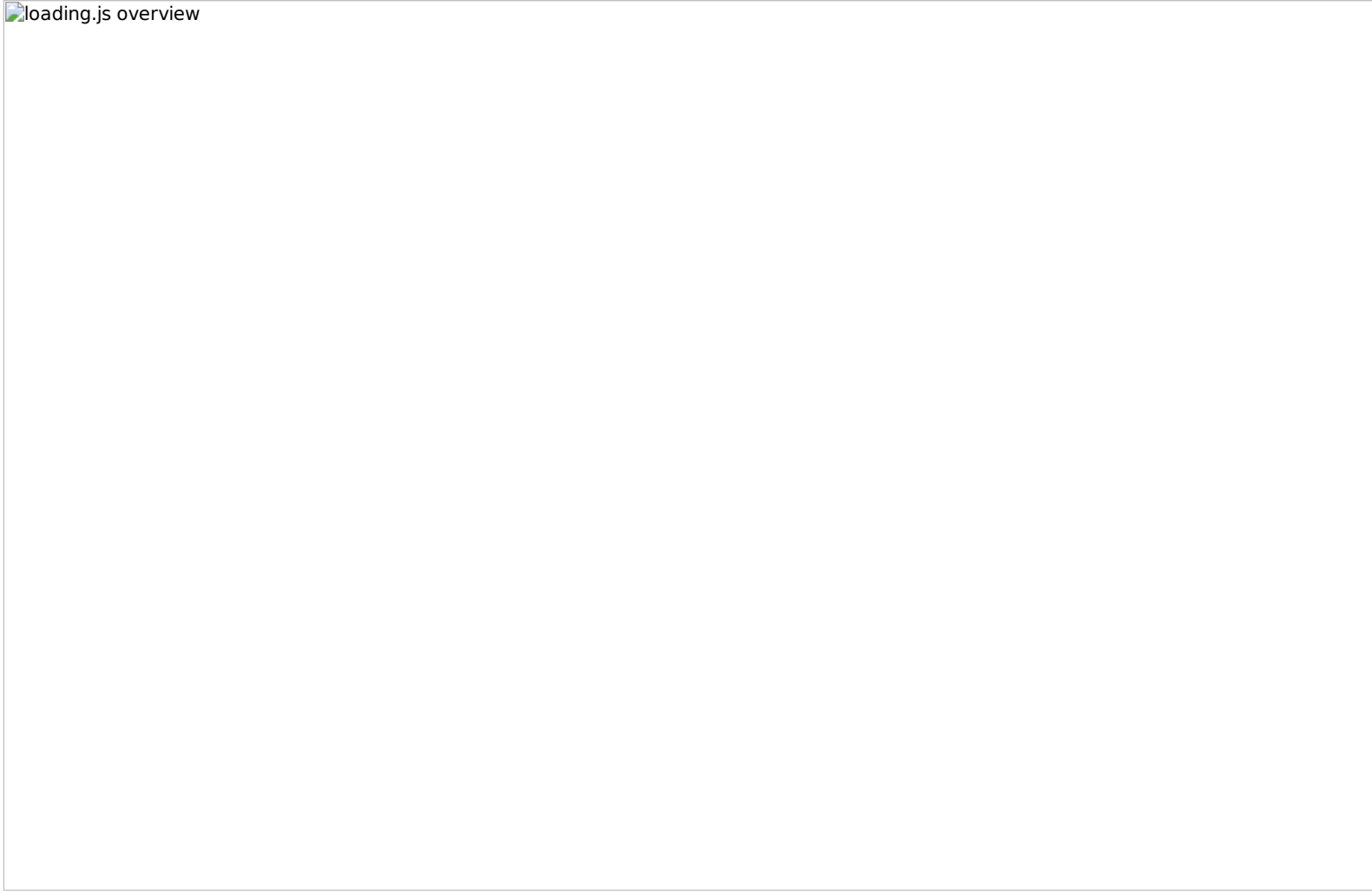
Create a loading state by adding a `loading.js` file inside a folder.




```
export default function Loading() {
  // You can add any UI inside Loading, including a Skeleton.
  return <LoadingSkeleton />
}
```

```
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.



Good to know:

- Navigation is immediate, even with server-centric routing.
- Navigation is interruptible, meaning changing routes does not need to wait for the content of the route to fully load before navigating to another route.
- Shared layouts remain interactive while new route segments load.

Recommendation: Use the `loading.js` convention for route segments (layouts and pages) as Next.js optimizes this functionality.

Streaming with Suspense

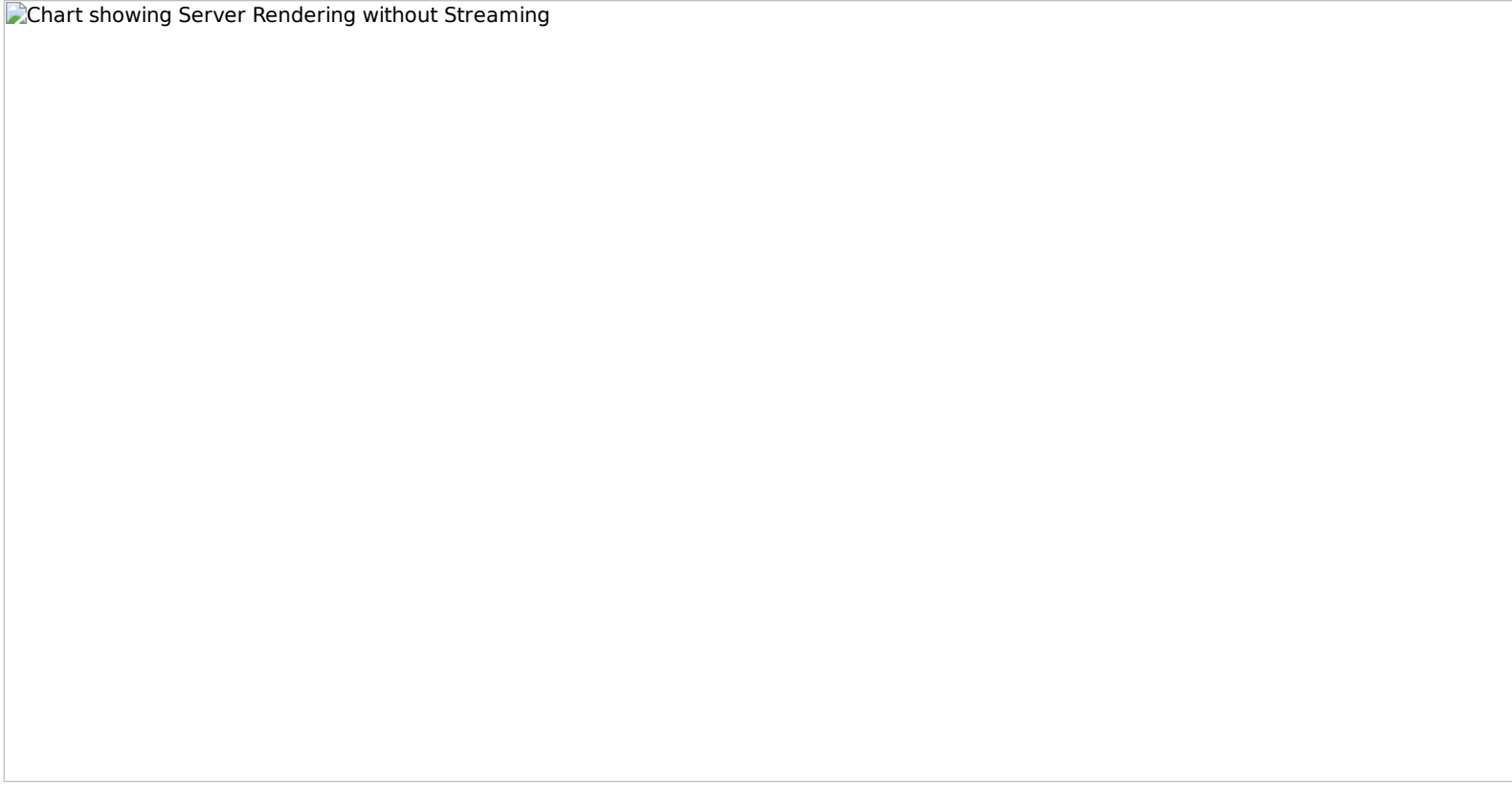
In addition to `loading.js`, you can also manually create Suspense Boundaries for your own UI components. The App Router supports streaming with Suspense for both Node.js and Edge runtimes.

What is Streaming?

To learn how Streaming works in React and Next.js, it's helpful to understand **Server-Side Rendering (SSR)** and its limitations.

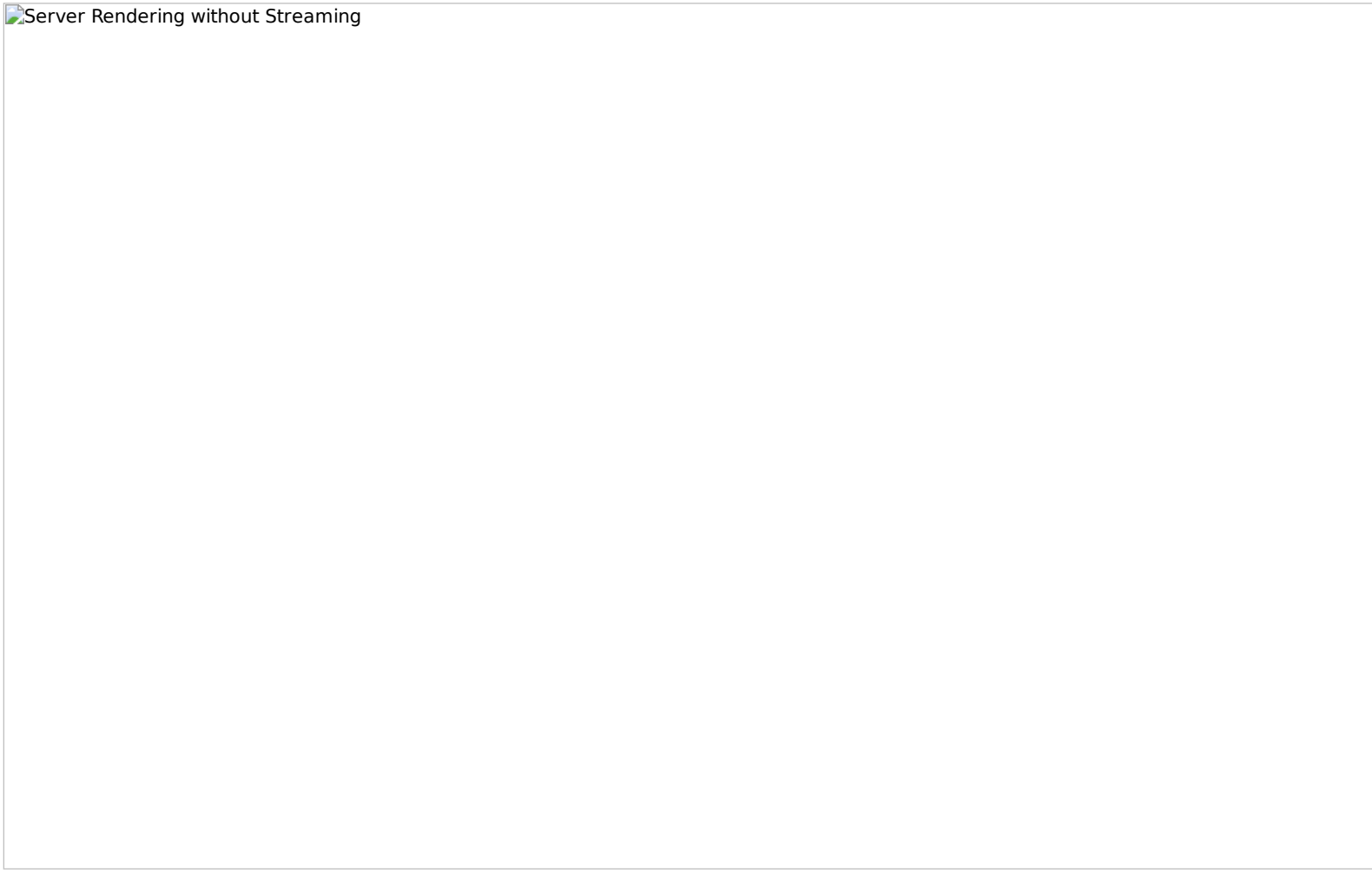
With SSR, there's a series of steps that need to be completed before a user can see and interact with a page:

1. First, all data for a given page is fetched on the server.
2. The server then renders the HTML for the page.
3. The HTML, CSS, and JavaScript for the page are sent to the client.
4. A non-interactive user interface is shown using the generated HTML, and CSS.
5. Finally, React hydrates the user interface to make it interactive.



These steps are sequential and blocking, meaning the server can only render the HTML for a page once all the data has been fetched. And, on the client, React can only hydrate the UI once the code for all components in the page has been downloaded.

SSR with React and Next.js helps improve the perceived loading performance by showing a non-interactive page to the user as soon as possible.



However, it can still be slow as all data fetching on server needs to be completed before the page can be shown to the user.

Streaming allows you to break down the page's HTML into smaller chunks and progressively send those chunks from the server to the client.

This enables parts of the page to be displayed sooner, without waiting for all the data to load before any UI can be rendered.

Streaming works well with React's component model because each component can be considered a chunk. Components that have higher priority (e.g. product information) or that don't rely on data can be sent first (e.g. layout), and React can start hydration earlier. Components that have lower priority (e.g. reviews, related products) can be sent in the same server request after their data has been fetched.

Streaming is particularly beneficial when you want to prevent long data requests from blocking the page from rendering as it can reduce the Time To First Byte (TTFB) and First Contentful Paint (FCP). It also helps improve Time to Interactive (TTI), especially on slower devices.

Example

`<Suspense>` works by wrapping a component that performs an asynchronous action (e.g. fetch data), showing fallback UI (e.g. skeleton, spinner) while it's happening, and then swapping in your component once the action completes.

```
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>
  )
}
```


```
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.

SEO

- Next.js will wait for data fetching inside `generateMetadata` to complete before streaming UI to the client. This guarantees the first part of a streamed response includes `<head>` tags.
- Since streaming is server-rendered, it does not impact SEO. You can use the Mobile Friendly Test tool from Google to see how your page appears to Google's web crawlers and view the serialized HTML (source).

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` or `notFound`. 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.

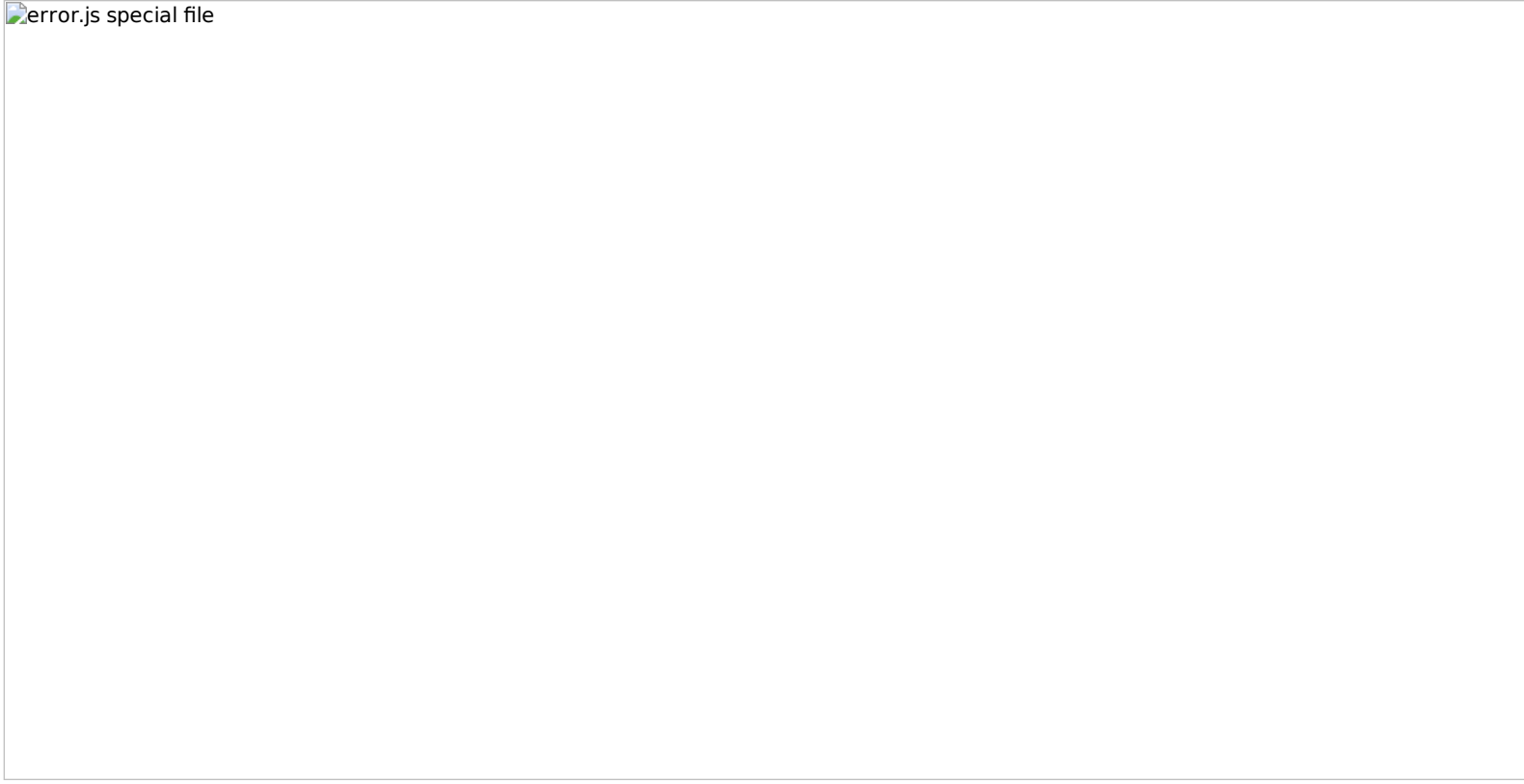
---

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.

- Automatically wrap a route segment and its nested children in a React Error Boundary.
- Create error UI tailored to specific segments using the file-system hierarchy to adjust granularity.
- Isolate errors to affected segments while keeping the rest of the application functional.
- Add functionality to attempt to recover from an error without a full page reload.

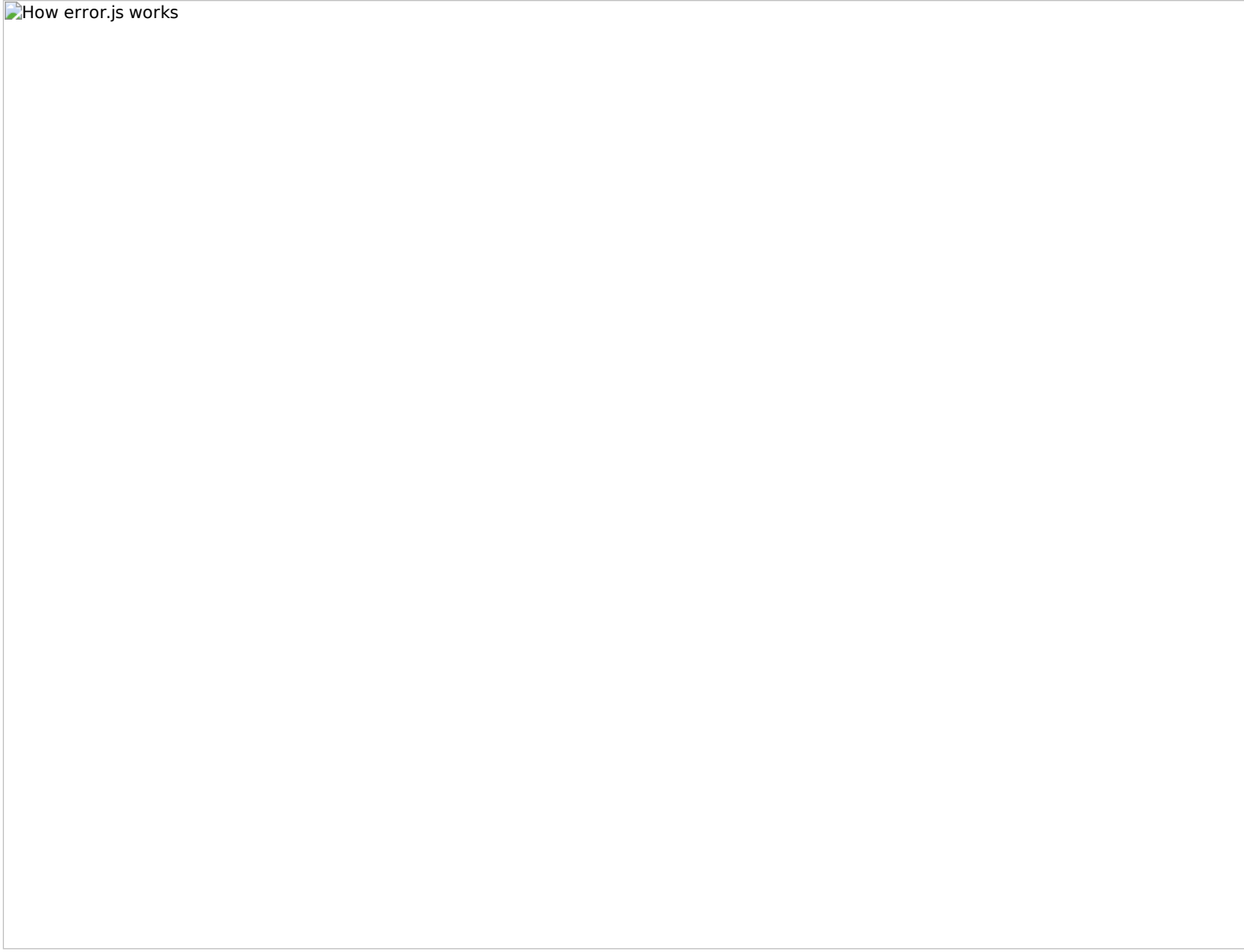
Create error UI by adding an `error.js` file inside a route segment and exporting a React component:




```
'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>
  )
}
```

```
'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



- `error.js` automatically creates a React Error Boundary that **wraps** a nested child segment or `page.js` component.
- The React component exported from the `error.js` file is used as the **fallback** component.
- If an error is thrown within the error boundary, the error is **contained**, and the fallback component is **rendered**.
- When the fallback error component is active, layouts **above** the error boundary **maintain** their state and **remain** interactive, and the error component can display functionality to recover from the error.

Recovering From Errors

The cause of an error can sometimes be temporary. In these cases, simply trying again might resolve the issue.

An error component can use the `reset()` function to prompt the user to attempt to recover from the error. When executed, the function will try to re-render the Error boundary's contents. If successful, the fallback error component is replaced with the result of the re-render.

```
'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>
  )
}
```

```
'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 are rendered in a specific nested hierarchy.

For example, a nested route with two segments that both include `layout.js` and `error.js` files are rendered in the following *simplified* component hierarchy:

The nested component hierarchy has implications for the behavior of `error.js` files across a nested route:

- Errors bubble up to the nearest parent error boundary. This means an `error.js` file will handle errors for all its nested child segments. More or less granular error UI can be achieved by placing `error.js` files at different levels in the nested folders of a route.
- An `error.js` boundary will **not** handle errors thrown in a `layout.js` component in the **same** segment because the error boundary is nested **inside** that 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 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.

```
'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>
  )
}
```

```
'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.

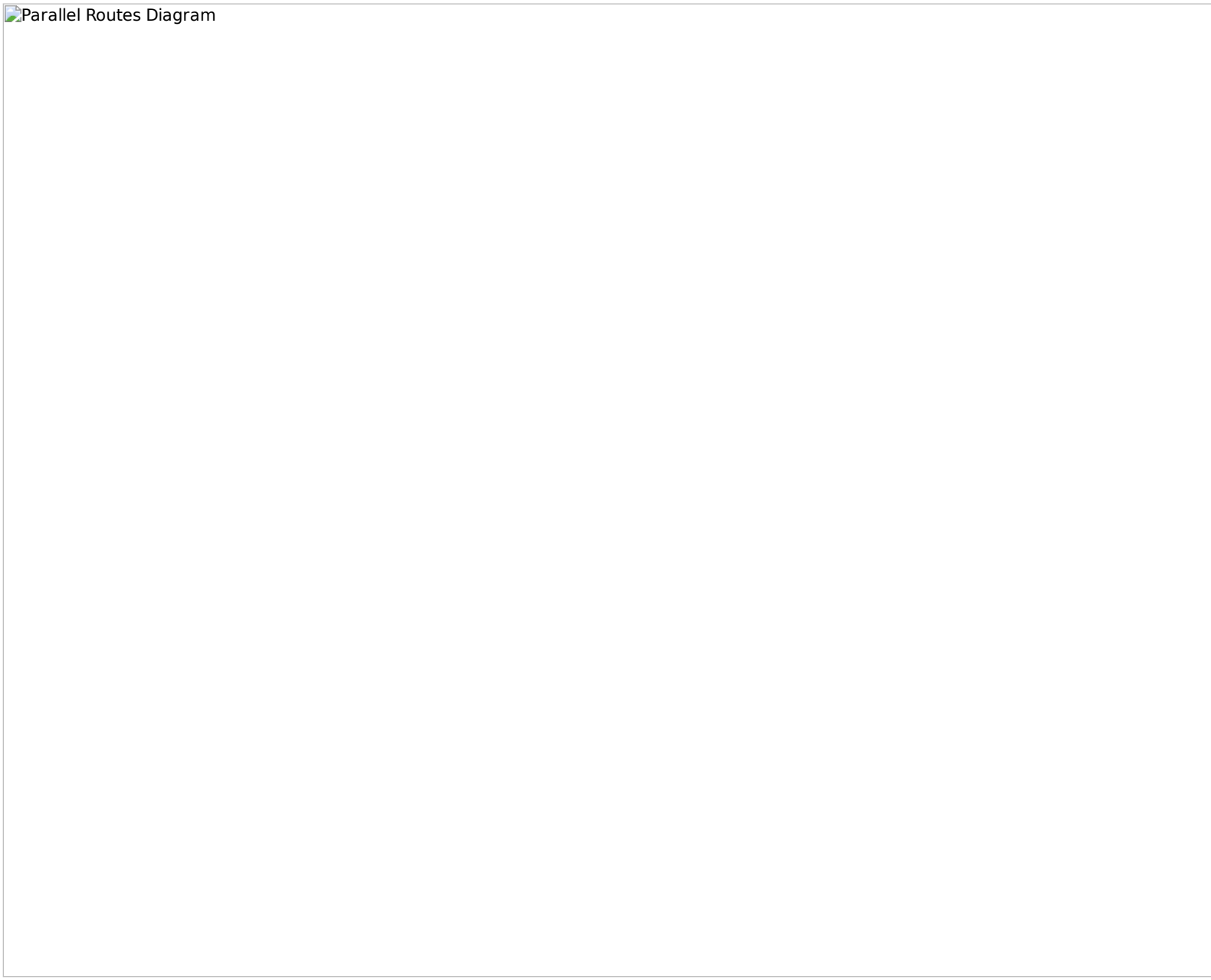
During development, the `Error` object forwarded to the client will be serialized and include the `message` of the original error for easier debugging.

---

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.

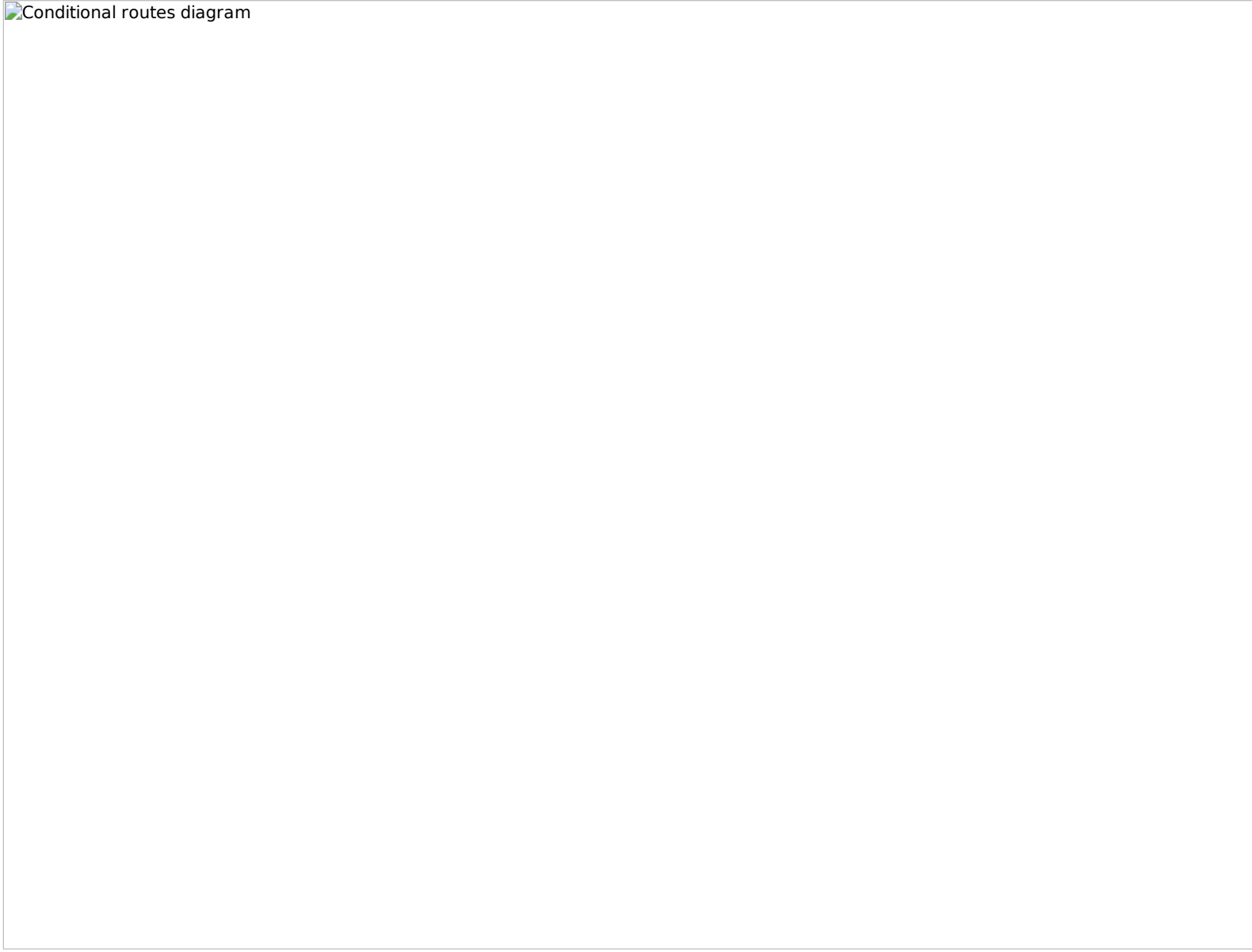
Parallel Routing allows you to simultaneously or conditionally render one or more pages in the same layout. For highly dynamic sections of an app, such as dashboards and feeds on social sites, Parallel Routing can be used to implement complex routing patterns.

For example, you can simultaneously render the team and analytics pages.



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

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.



Convention

Parallel routes are created using named **slots**. Slots are defined with the `@folder` convention, and are passed to the same-level layout as props.

> Slots are *not* route segments and *do not affect the URL structure*. The file path `/@team/members` would be accessible at `/members`.

For example, the following file structure defines two explicit slots: `@analytics` and `@team`.

The folder structure above means that the component in `app/layout.js` now accepts the `@analytics` and `@team` slots props, and can render them in parallel alongside the `children` prop:

```
export default function Layout(props: {
  children: React.ReactNode
  analytics: React.ReactNode
  team: React.ReactNode
}) {
  return (
    <>
      {props.children}
      {props.team}
      {props.analytics}
    </>
  )
}
```

```
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

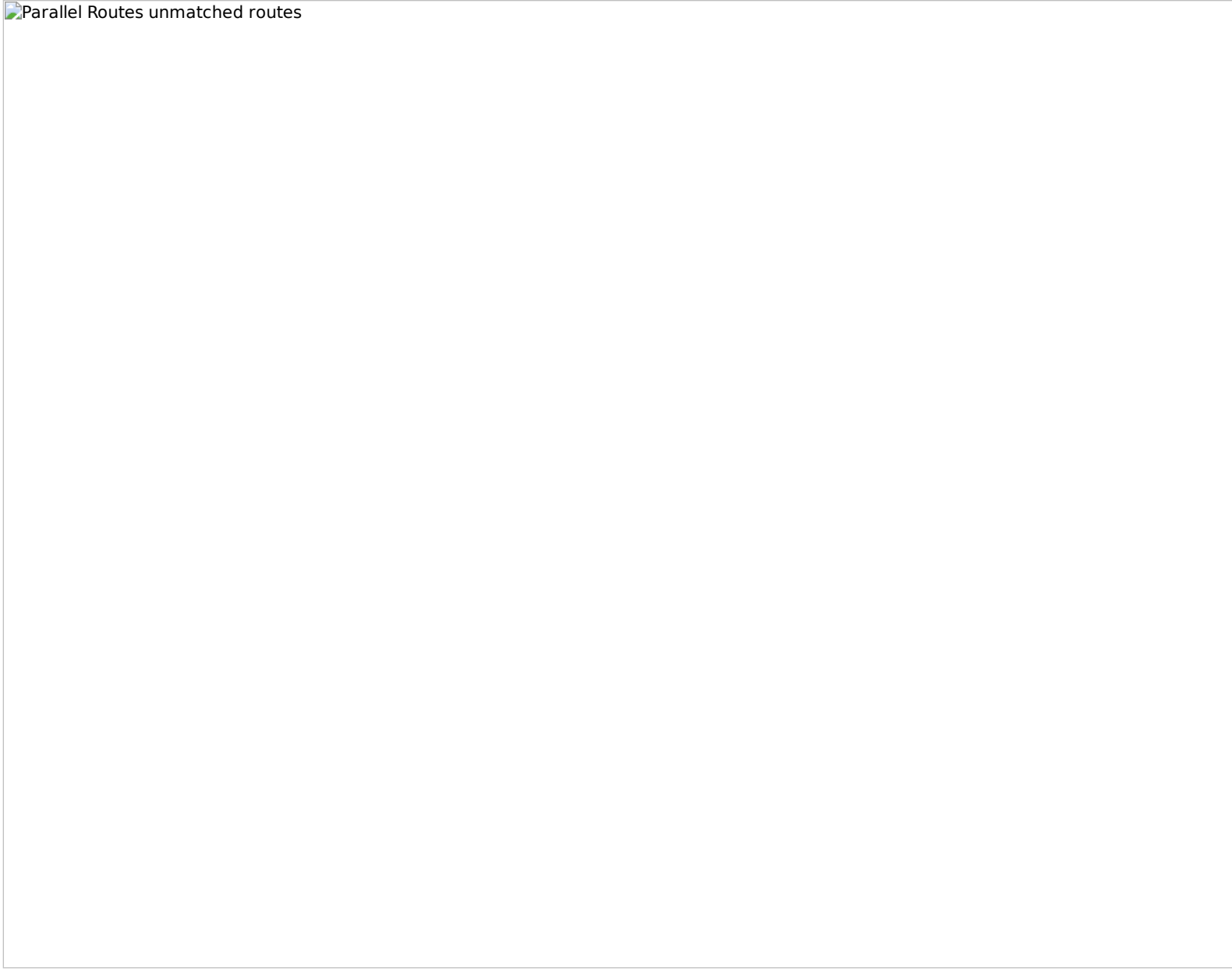
By default, the content rendered within a slot will match the current URL.

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.



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` and `useSelectedLayoutSegments` accept a `parallelRoutesKey`, which allows you to read the active route segment within that slot.

```
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

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

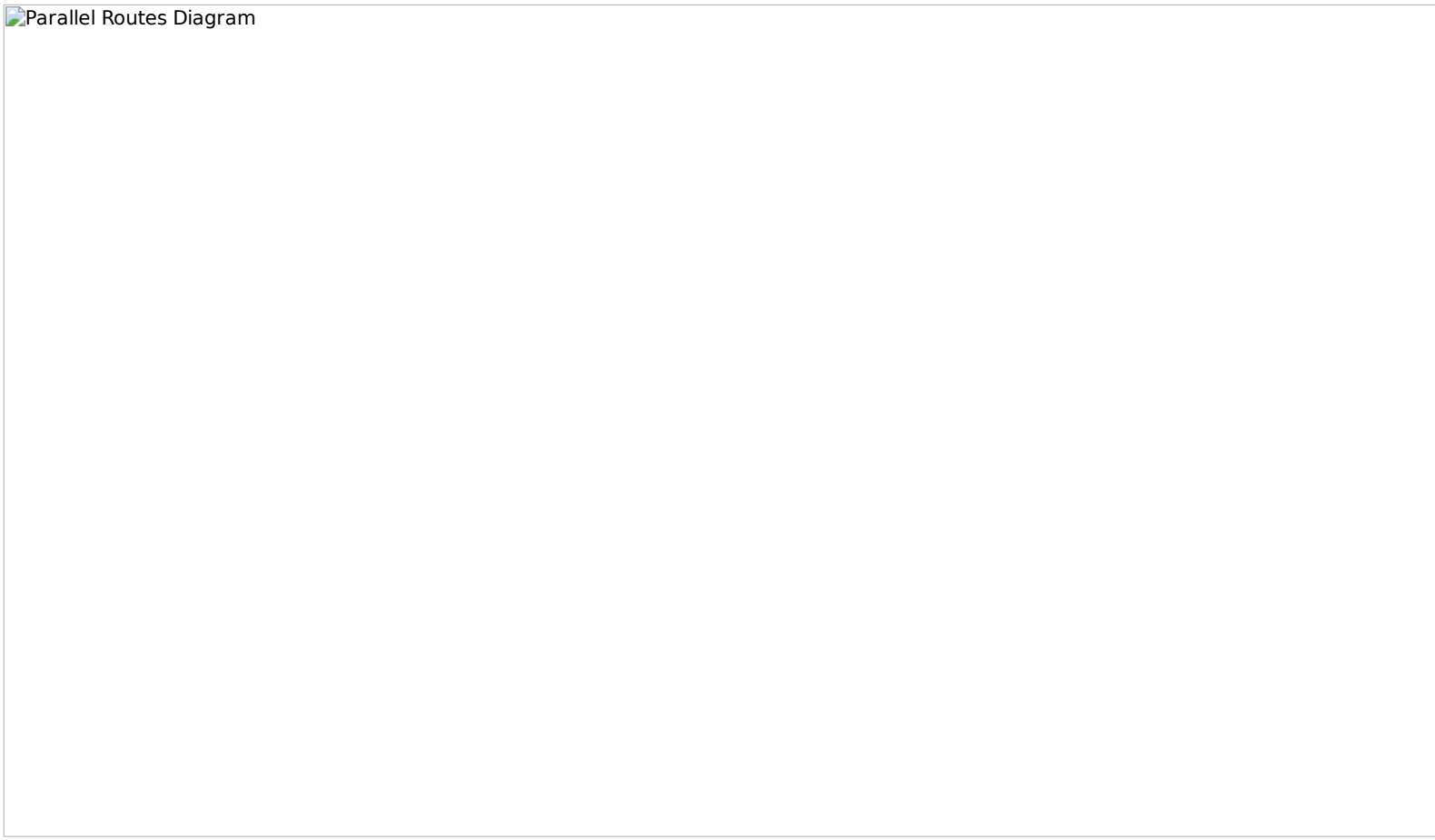
```
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

export default 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.



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

```
export default async function Layout(props: {
  // ...
  auth: React.ReactNode
}) {
  return (
    <>
      {/* ... */}
      {props.auth}
    </>
  )
}
```

```
export default async function Layout(props) {
  return (
    <>
      {/* ... */}
      {props.auth}
    </>
  )
}
```

```
import { Modal } from 'components/modal'

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

```
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`.

```
export default function Default() {
  return null
}
```

```
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.

```
'use client'
import { useRouter } from 'next/navigation'
import { Modal } from 'components/modal'

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

```
'use client'
import { useRouter } from 'next/navigation'
import { Modal } from 'components/modal'

export default 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 section.

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


```
export default function CatchAll() {
  return null
}
```

```
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.

```
import { getUser } from '@/lib/auth'

export default function Layout({
  dashboard,
  login,
}: {
  dashboard: React.ReactNode
  login: React.ReactNode
```

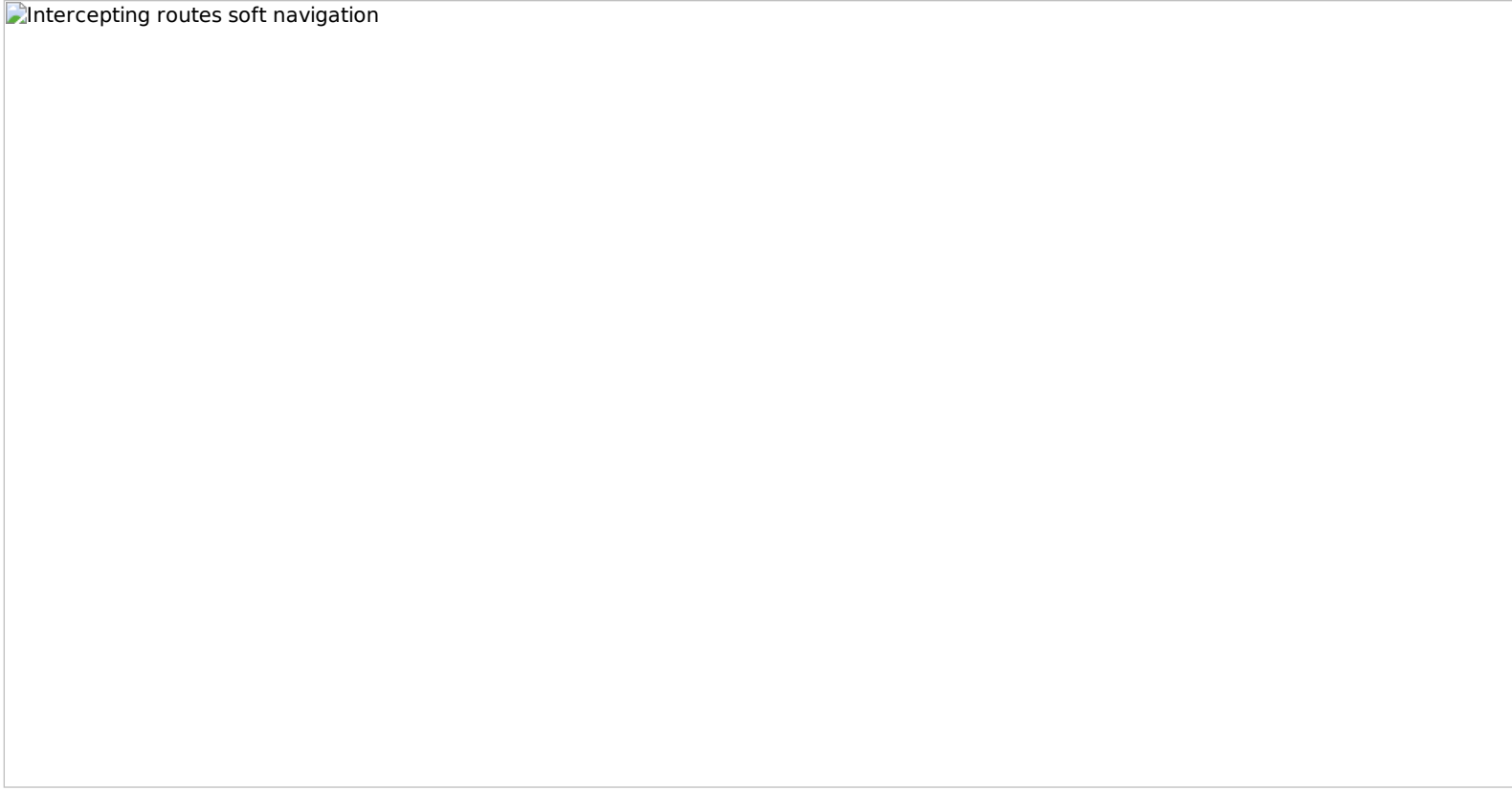
```
}) {
  const isLoggedIn = getUser()
  return isLoggedIn ? dashboard : login
}

import { getUser } from '@/lib/auth'

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

---

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`.



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.

Convention

Intercepting routes can be defined with the `(..)` convention, which is similar to relative path convention `../` but for segments.

You can use:

- `(.)` to match segments on the **same level**
- `(..)` to match segments **one level above**
- `(..)(..)` to match segments **two levels above**
- `(...)` to match segments from the **root** `app` directory

For example, you can intercept the `photo` segment from within the `feed` segment by creating a `(..)photo` directory.

Note that the `(..)` convention is based on *route segments*, not the file-system.

Examples

Modals

Intercepting Routes can be used together with Parallel Routes to create modals.

Using this pattern to create modals overcomes some common challenges when working with modals, by allowing you to:

- Make the modal content **shareable through a URL**
- **Preserve context** when the page is refreshed, instead of closing the modal
- **Close the modal on backwards navigation** rather than going to the previous route
- **Reopen the modal on forwards navigation**

> In the above example, the path to the `photo` segment can use the `(..)` matcher since `@modal` is a *slot* and not a *segment*. This means that the `photo` route is only one *segment* level higher, despite being two *file-system* levels higher.

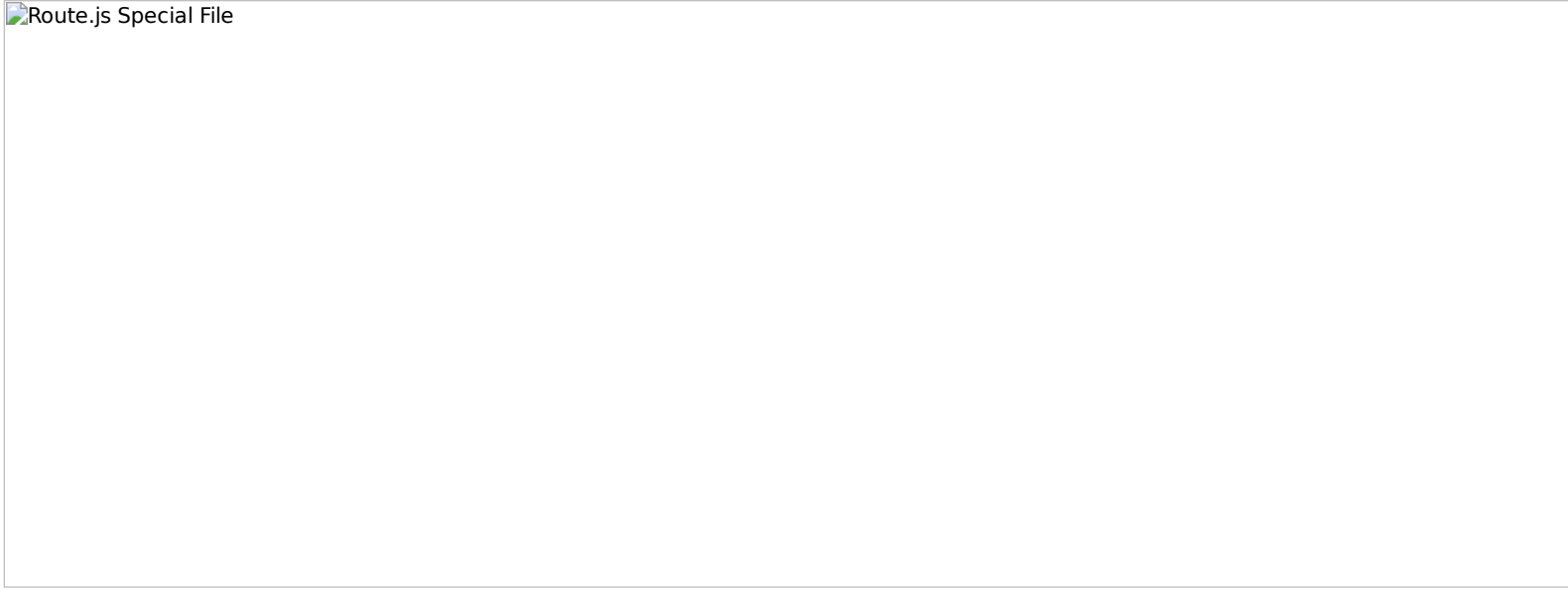
Other examples could include opening a login modal in a top navbar while also having a dedicated `/login` page, or opening a shopping cart in a side modal.

View an example of modals with Intercepted and Parallel Routes.

---

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 and Response APIs.



> **Good to know**: Route Handlers are only available inside the `app` directory. They are the equivalent of API Routes inside the `pages` directory meaning you **do not** need to use API Routes and Route Handlers together.

Convention

Route Handlers are defined in a `route.js|ts` file inside the `app` directory:

```
export const dynamic = 'force-dynamic' // defaults to auto
export async function GET(request: Request) {}
```

```
export const dynamic = 'force-dynamic' // defaults to auto
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 are supported: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, and `OPTIONS`. If an unsupported method is called, Next.js will return a `405 Method Not Allowed` response.

Extended `NextRequest` and `NextResponse` APIs

In addition to supporting native Request and Response. Next.js extends them with `NextRequest` and `NextResponse` to provide convenient helpers for advanced use cases.

Behavior

Caching

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

```
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 })
}
```

```
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()` 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 like `cookies` and `headers`.
- The Segment Config Options manually specifies dynamic mode.

For example:

```
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 })
}
```

```
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.

```
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)
}
```

```
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 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`	Conflict
`app/page.js`	`app/api/route.js`	Valid
`app/[user]/page.js`	`app/api/route.js`	Valid

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

```
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 using the `next.revalidate` option:

```
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)
}
```

```
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]:

```
export const revalidate = 60
```

Dynamic Functions

Route Handlers can be used with dynamic functions from Next.js, like [`cookies`] and [`headers`].

Cookies

You can read or set cookies with [`cookies`] from `next/headers`. This server function can be called directly in a Route Handler, or nested inside of another function.

Alternatively, you can return a new `Response` using the [`Set-Cookie`] header.

```
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}` },
  })
}
```

```
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}` },
  })
}
```

You can also use the underlying Web APIs to read cookies from the request ([`NextRequest`]):

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

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

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

Headers

You can read headers with [`headers`] from `next/headers`. This server function can be called directly in a Route Handler, or nested inside of another function.

This `headers` instance is read-only. To set headers, you need to return a new `Response` with new `headers`.

```
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 },
  })
}
```

```
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 },
  })
}
```

You can also use the underlying Web APIs to read headers from the request ([`NextRequest`]):

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

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

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

Redirects

```
import { redirect } from 'next/navigation'
```

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

```
import { redirect } from 'next/navigation'

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

Dynamic Route Segments

> We recommend reading the Defining Routes page before continuing.

Route Handlers can use Dynamic Segments to create request handlers from dynamic data.

```
export async function GET(
  request: Request,
  { params }: { params: { slug: string } }
) {
  const slug = params.slug // 'a', 'b', or 'c'
}
```

```
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, including for more easily handling query parameters.

```
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
}
```

```
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.

```
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: 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)
}
```

```
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.

```
// 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)
}

// 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()

      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:

```
export async function POST(request: Request) {
  const res = await request.json()
  return Response.json({ res })
}

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:

```
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 })
}

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`] 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:

```
export const dynamic = 'force-dynamic' // defaults to auto

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',
    },
  })
}
```

```
export const dynamic = 'force-dynamic' // defaults to auto

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',
    },
  })
}
```

Webhooks

You can use a Route Handler to receive webhooks from third-party services:

```
export async function POST(request: Request) {
  try {
    const text = await request.text()
    // Process the webhook payload
  } catch (error) {
    return new Response(`Webhook error: ${error.message}`, {
      status: 400,
    })
  }

  return new Response('Success!', {
    status: 200,
  })
}
```

```
export async function POST(request) {
  try {
    const text = await request.text()
    // Process the webhook payload
  } catch (error) {
    return new Response(`Webhook error: ${error.message}`, {
      status: 400,
    })
  }

  return new Response('Success!', {
    status: 200,
  })
}
```

Notably, unlike API Routes with the Pages Router, you do not need to use `bodyParser` to use any additional configuration.

Edge and Node.js Runtimes

Route Handlers have an isomorphic Web API to support both Edge and Node.js runtimes seamlessly, including support for streaming. Since Route Handlers use the same route segment configuration as Pages and Layouts, they support long-awaited features like general-purpose statically regenerated Route Handlers.

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

```
export const runtime = 'edge' // 'nodejs' is the default
```

Non-UI Responses

You can use Route Handlers to return non-UI content. Note that `sitemap.xml`, `robots.txt`, `app icons`, and open graph images all have built-in support.

```
export const dynamic = 'force-dynamic' // defaults to auto

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>`,
    {
      headers: {
        'Content-Type': 'text/xml',
      },
    }
  )
}
```

```
export const dynamic = 'force-dynamic' // defaults to auto

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 as pages and layouts.

```
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 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 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 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

```
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*',
}
```

```
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
2. Conditional statements

Matcher

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

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

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

```
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:

```
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).*)',
  ],
}
```

You can also ignore prefetches (from `next/link`) that don't need to go through the Middleware using the `missing` array:

```
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' },
      ],
    },
  ],
}
```

> **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 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

```
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))
  }
}
```

```
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

To produce a response from Middleware, you can:

1. `rewrite` to a route (Page or Route Handler) that produces a response
2. return a `NextResponse` directly. See Producing a Response

To produce a response from Middleware, you can:

1. `rewrite` to a route (Page or Edge API Route) that produces a response
2. return a `NextResponse` directly. See Producing a Response

Using Cookies

Cookies are regular headers. On a `Request`, they are stored in the `Cookie` header. On a `Response` they are in the `Set-Cookie` header. Next.js provides a convenient way to access and manipulate these cookies through the `cookies` extension on `NextRequest` and `NextResponse`.

1. For incoming requests, `cookies` comes with the following methods: `get`, `getAll`, `set`, and `delete` cookies. You can check for the existence of a cookie with `has` or remove all cookies with `clear`.
2. For outgoing responses, `cookies` have the following methods `get`, `getAll`, `set`, and `delete`.

```
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=/` header.

  return response
}
```

```
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).

```
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
}
```

```
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 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)

```
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 }
    )
  }
}
```

```
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 }
    )
  }
}
```

`waitUntil` and `NextFetchEvent`

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

The `waitUntil()` method takes a promise as an argument, and extends the lifetime of the Middleware until the promise settles. This is useful for performing work in the background.

```
import { NextResponse } from 'next/server'
import type { NextFetchEvent, NextRequest } from 'next/server'

export function middleware(req: NextRequest, event: NextFetchEvent) {
  event.waitUntil(
    fetch('https://my-analytics-platform.com', {
      method: 'POST',
      body: JSON.stringify({ pathname: req.nextUrl.pathname }),
    })
  )

  return NextResponse.next()
}
```

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` disables Next.js redirects for adding or removing trailing slashes. This allows custom handling inside middleware to maintain the trailing slash for some paths but not others, which can make incremental migrations easier.

```
module.exports = {
  skipTrailingSlashRedirect: true,
}

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.

```
module.exports = {
  skipMiddlewareUrlNormalize: true,
}

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. 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
`v12.0.9`	Enforce absolute URLs in Edge Runtime (PR)
`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, Next.js is **unopinionated** about how you organize and colocate your project files.

This page shares default behavior and features you can use to organize your project.

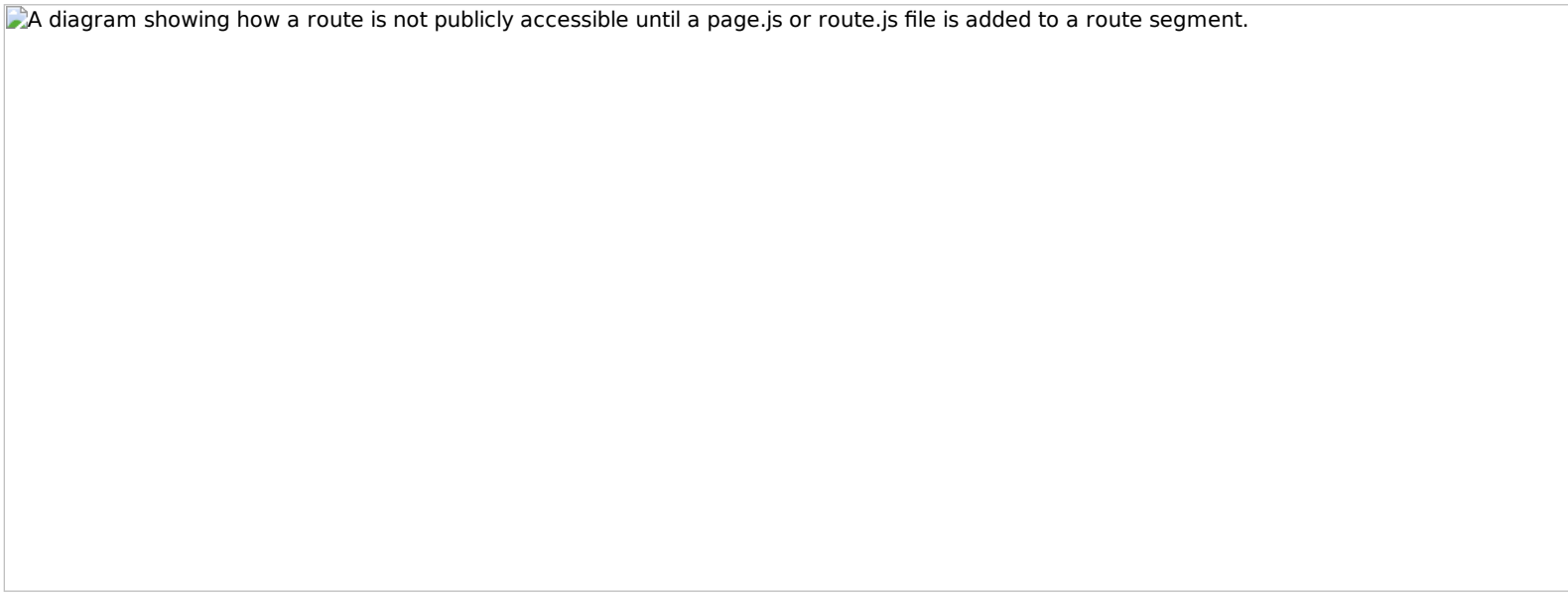
- Safe colocation by default
- Project organization features
- Project organization strategies

Safe colocation by default

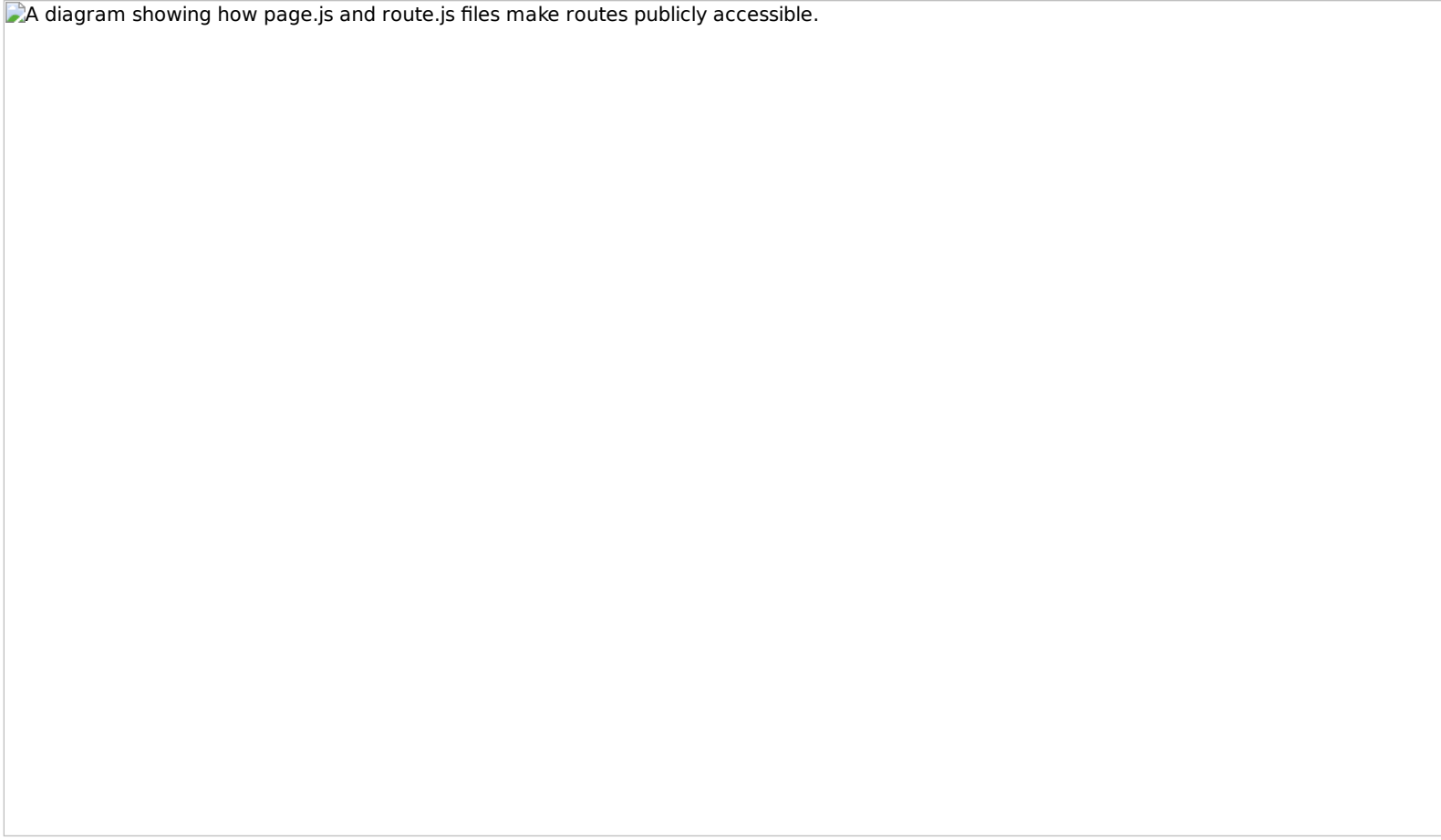
In the `app` directory, nested folder hierarchy defines route structure.

Each folder represents a route segment that is mapped to a corresponding segment in a URL path.

However, even though route structure is defined through folders, a route is **not publicly accessible** until a `page.js` or `route.js` file is added to a route segment.



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



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

Good to know:

- This is different from the `pages` directory, where any file in `pages` is considered a route.
- While you **can** colocate your project files in `app` you don't **have** to. If you prefer, you can keep them outside the `app` directory.

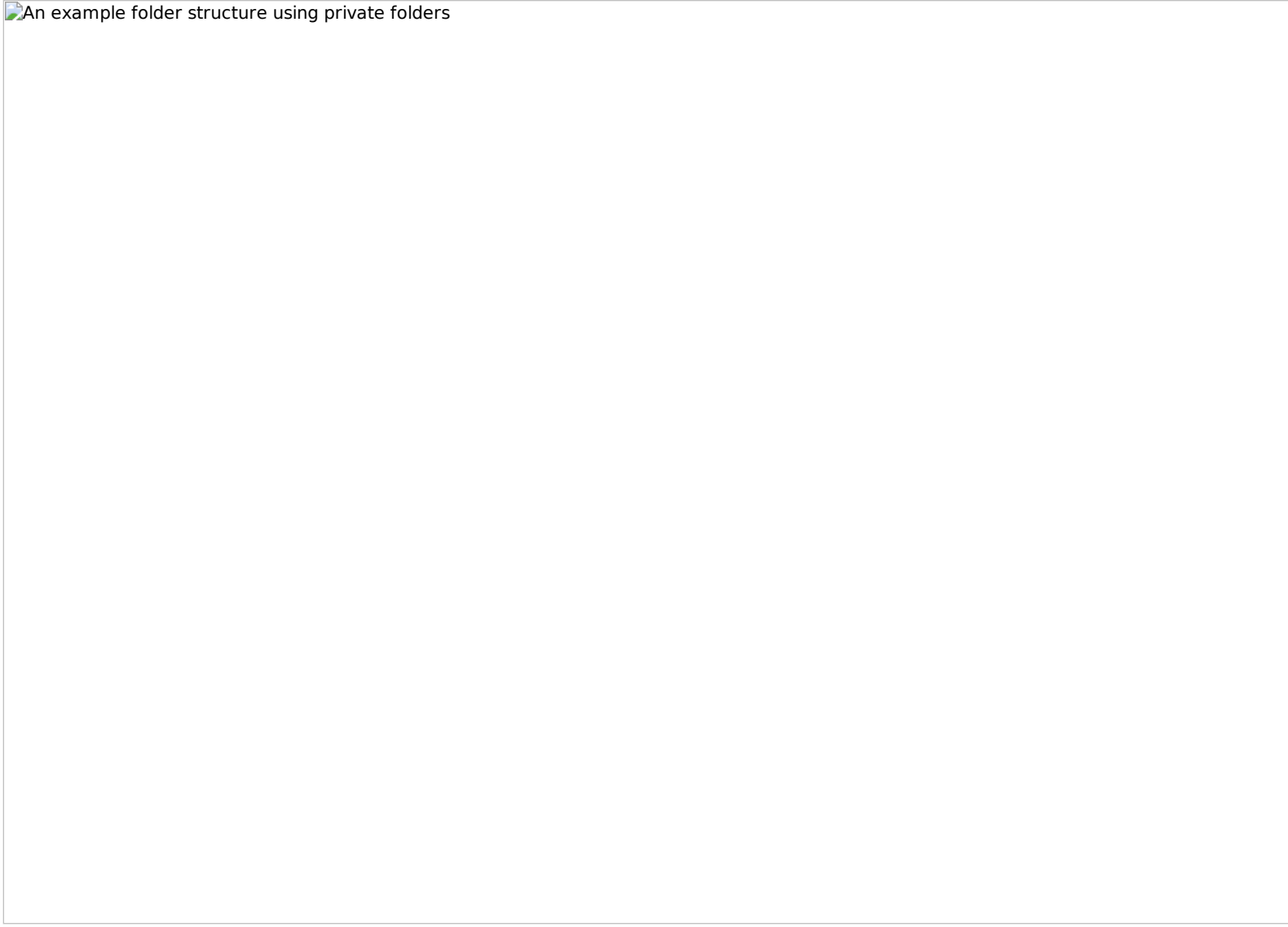
Project organization features

Next.js provides several features to help you organize your project.

Private Folders

Private folders can be created by prefixing a folder with an underscore: `_folderName`

This indicates the folder is a private implementation detail and should not be considered by the routing system, thereby **opting the folder and all its subfolders** out of routing.



Since files in the `app` directory can be safely colocated by default, private folders are not required for colocation. However, they can be useful for:

- Separating UI logic from routing logic.
- Consistently organizing internal files across a project and the Next.js ecosystem.
- Sorting and grouping files in code editors.
- Avoiding potential naming conflicts with future Next.js file conventions.

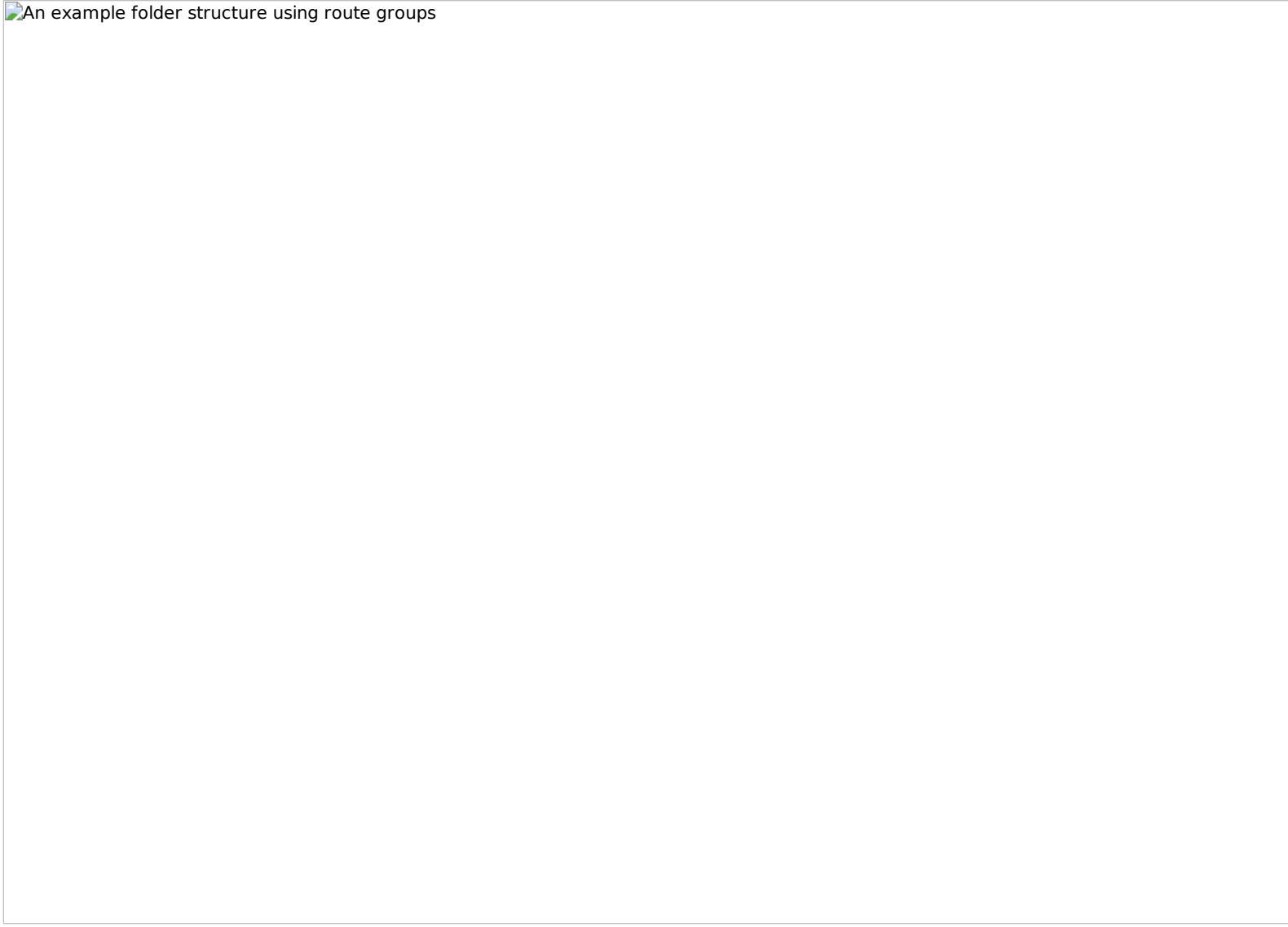
Good to know

- While not a framework convention, you might also consider marking files outside private folders as "private" using the same underscore pattern.
- You can create URL segments that start with an underscore by prefixing the folder name with `%5F` (the URL-encoded form of an underscore): `%5FfolderName`.
- If you don't use private folders, it would be helpful to know Next.js special file conventions to prevent unexpected naming conflicts.

Route Groups

Route groups can be created by wrapping a folder in parenthesis: `(folderName)`

This indicates the folder is for organizational purposes and should **not be included** in the route's URL path.



Route groups are useful for:

- Organizing routes into groups e.g. by site section, intent, or team.
- Enabling nested layouts in the same route segment level:
  - Creating multiple nested layouts in the same segment, including multiple root layouts
  - Adding a layout to a subset of routes in a common segment

`src` Directory

Next.js supports storing application code (including `app`) inside an optional `src` directory. This separates application code from project configuration files which mostly live in the root of a project.

Module Path Aliases

Next.js supports Module Path Aliases which make it easier to read and maintain imports across deeply nested project files.

```
// before
import { Button } from '../../../components/button'

// after
import { Button } from '@/components/button'
```

Project organization strategies

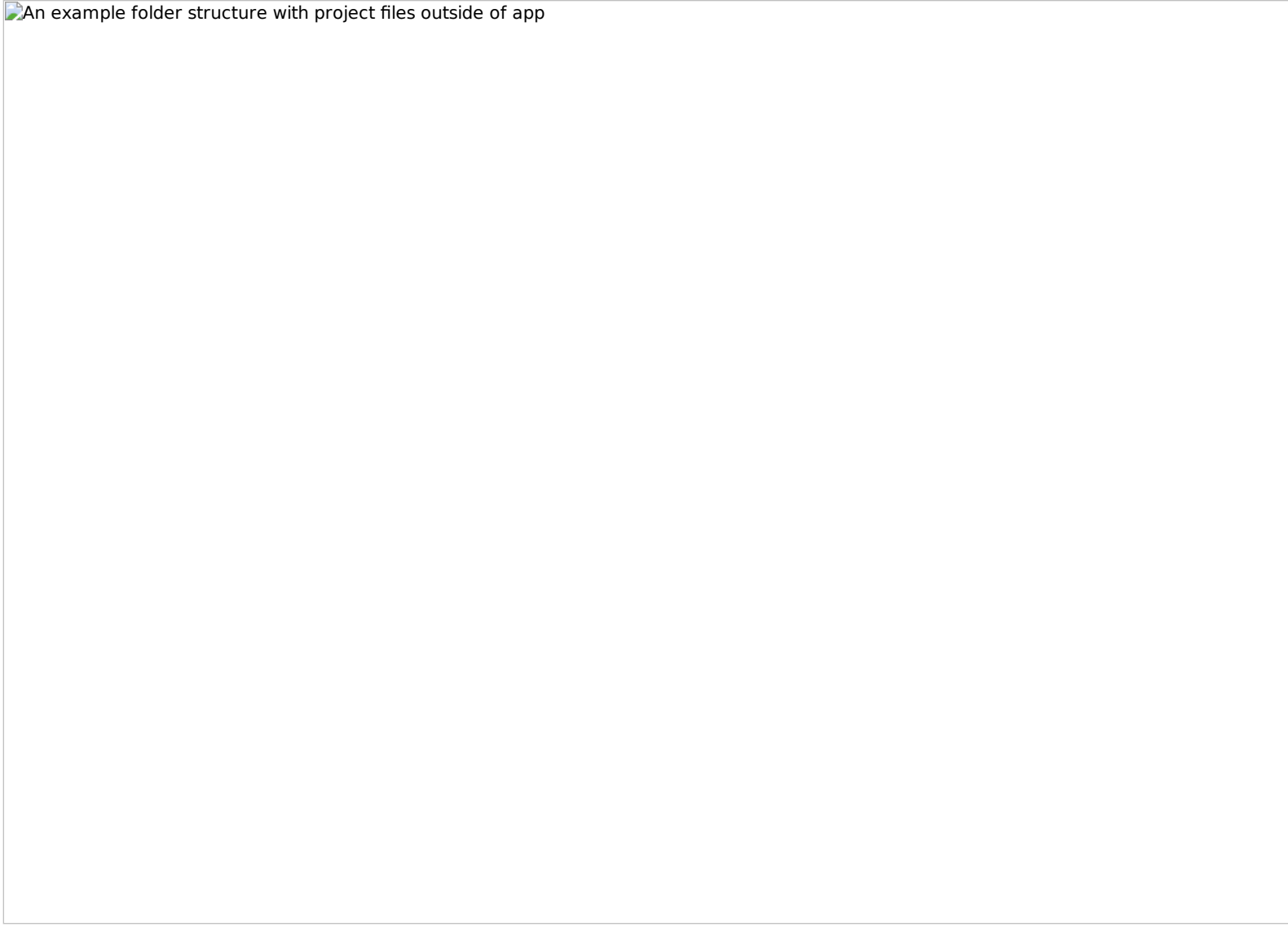
There is no "right" or "wrong" way when it comes to organizing your own files and folders in a Next.js project.

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.

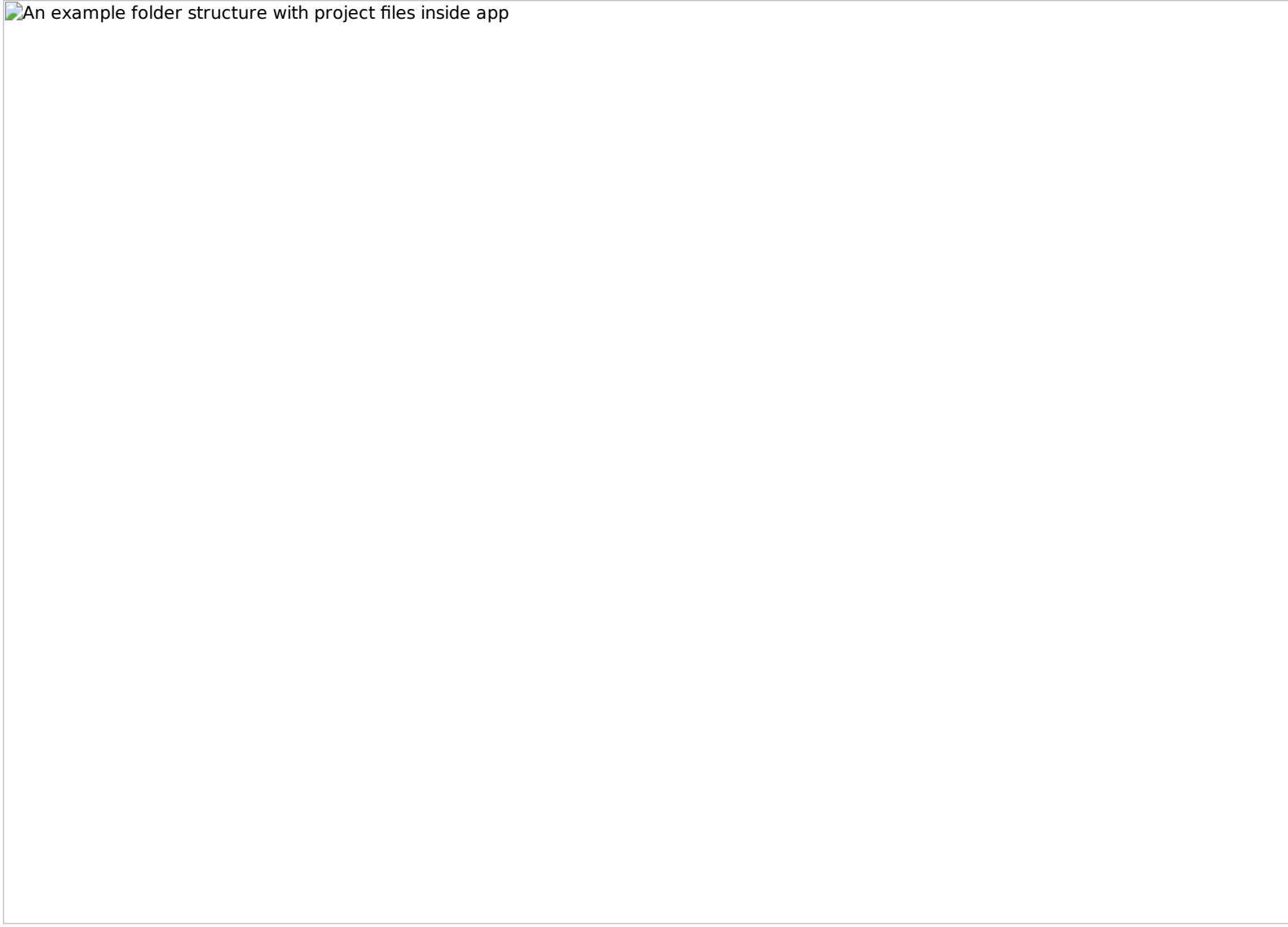
Store project files outside of `app`

This strategy stores all application code in shared folders in the **root of your project** and keeps the `app` directory purely for routing purposes.



Store project files in top-level folders inside of `app`

This strategy stores all application code in shared folders in the **root of the `app` directory**.



Split project files by feature or route

This strategy stores globally shared application code in the root `app` directory and **splits** more specific application code into the route segments that use them.

---

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.

```
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.

```
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:

```
// 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:

```
{
  "products": {
    "cart": "Add to Cart"
  }
}
```

```
{
  "products": {
    "cart": "Toevoegen aan Winkelwagen"
  }
}
```

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

```
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.

```
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, 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:

```
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
- `next-intl`
- `next-international`
- `next-i18n-router`

---

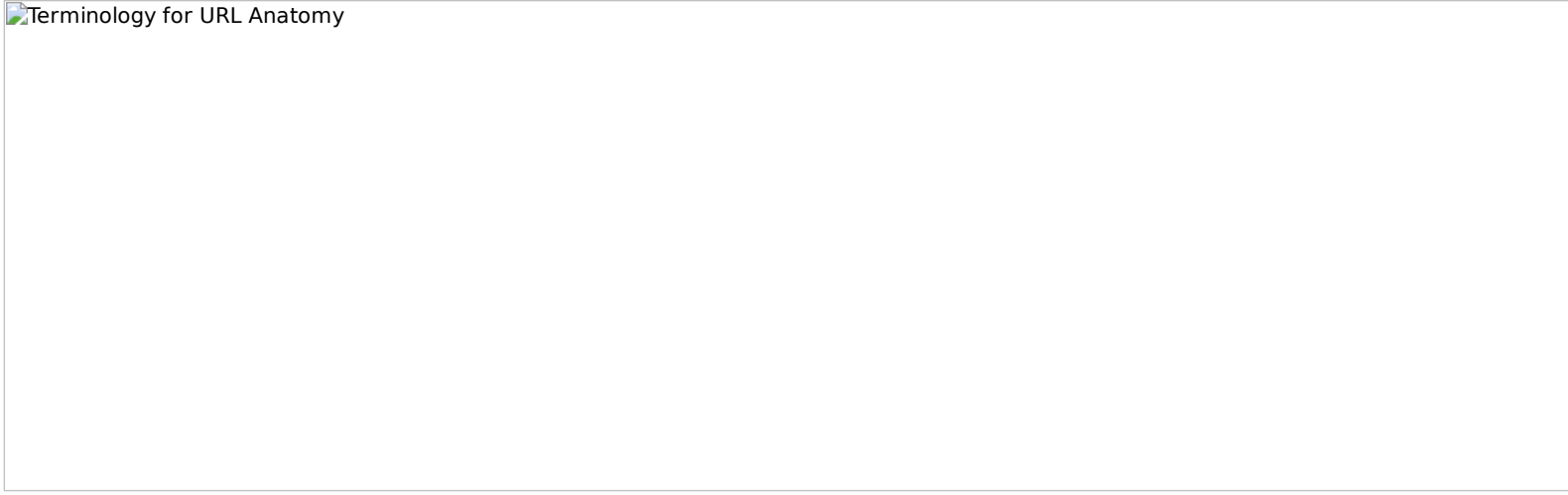
title: Routing Fundamentals nav_title: Routing description: Learn the fundamentals of routing for front-end applications.

The skeleton of every application is routing. This page will introduce you to the **fundamental concepts** of routing for the web and how to handle routing in Next.js.

Terminology

First, you will see these terms being used throughout the documentation. Here's a quick reference:

- **Tree:** A convention for visualizing a hierarchical structure. For example, a component tree with parent and children components, a folder structure, etc.
- **Subtree:** Part of a tree, starting at a new root (first) and ending at the leaves (last).
- **Root**: The first node in a tree or subtree, such as a root layout.
- **Leaf:** Nodes in a subtree that have no children, such as the last segment in a URL path.



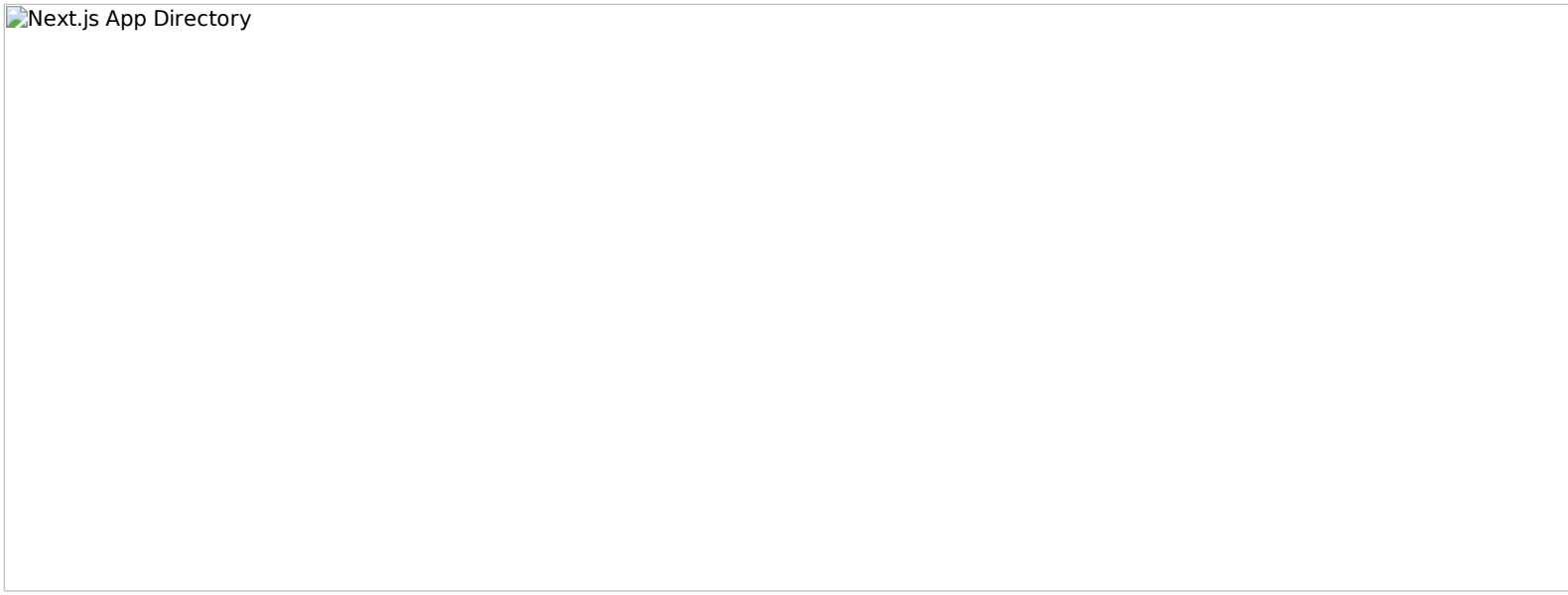
- **URL Segment:** Part of the URL path delimited by slashes.
- **URL Path:** Part of the URL that comes after the domain (composed of segments).

The `app` Router

In version 13, Next.js introduced a new **App Router** built on React Server Components, which supports shared layouts, nested routing, loading states, error handling, and more.

The App Router works in a new directory named `app`. The `app` directory works alongside the `pages` directory to allow for incremental adoption. This allows you to opt some routes of your application into the new behavior while keeping other routes in the `pages` directory for previous behavior. If your application uses the `pages` directory, please also see the Pages Router documentation.

> **Good to know**: The App Router takes priority over the Pages Router. Routes across directories should not resolve to the same URL path and will cause a build-time error to prevent a conflict.



By default, components inside `app` are React Server Components. This is a performance optimization and allows you to easily adopt them, and you can also use Client Components.

> **Recommendation:** Check out the Server page if you're new to Server Components.

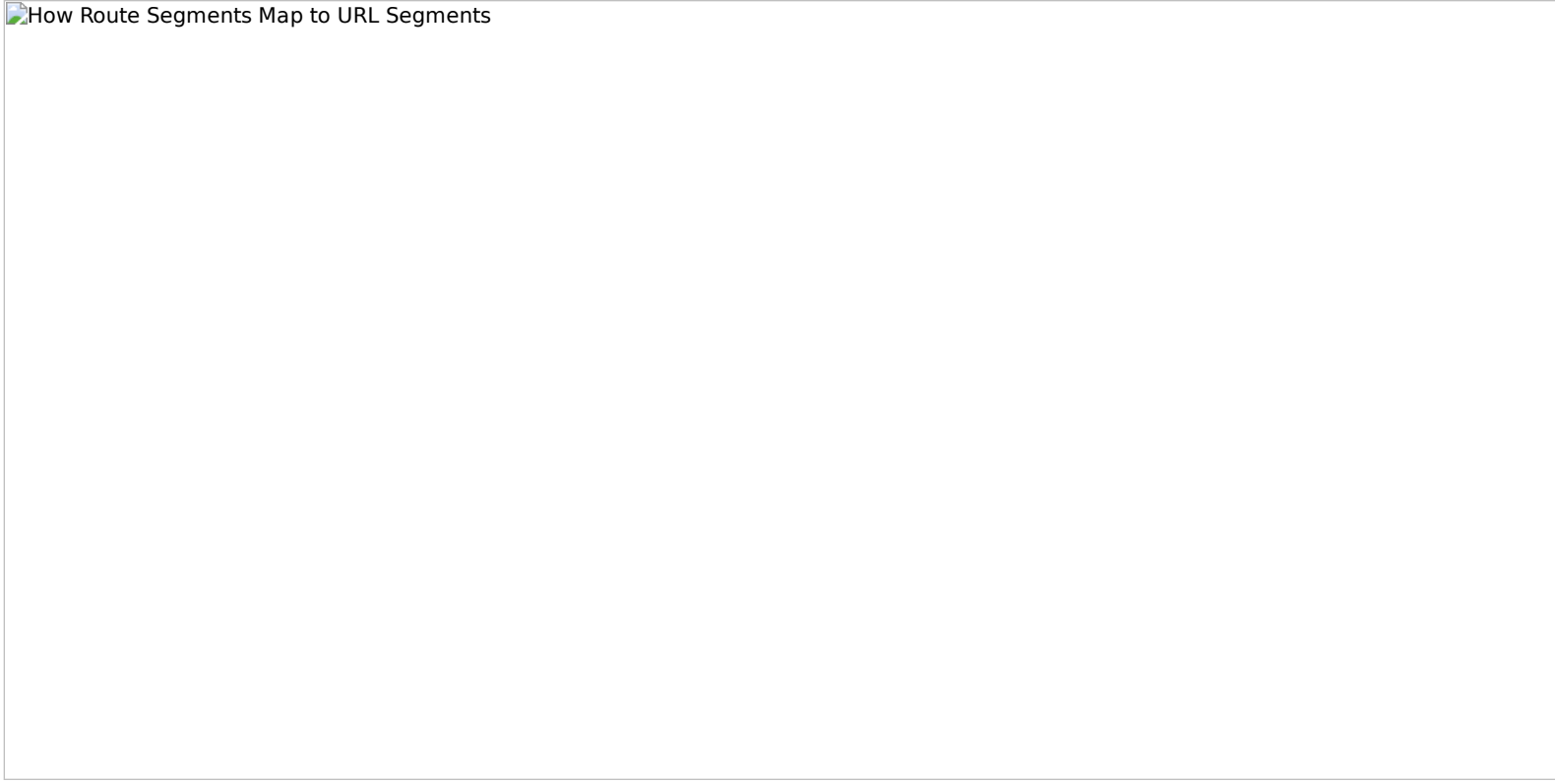
Roles of Folders and Files

Next.js uses a file-system based router where:

- **Folders** are used to define routes. A route is a single path of nested folders, following the file-system hierarchy from the **root folder** down to a final **leaf folder** that includes a `page.js` file. See Defining Routes.
- **Files** are used to create UI that is shown for a route segment. See special files.

Route Segments

Each folder in a route represents a **route segment**. Each route segment is mapped to a corresponding **segment** in a **URL path**.



Nested Routes

To create a nested route, you can nest folders inside each other. For example, you can add a new `/dashboard/settings` route by nesting two new folders in the `app` directory.

The `/dashboard/settings` route is composed of three segments:

- `/` (Root segment)
- `dashboard` (Segment)
- `settings` (Leaf segment)

File Conventions

Next.js provides a set of special files to create UI with specific behavior in nested routes:

`layout`	Shared UI for a segment and its children
`page`	Unique UI of a route and make routes publicly accessible
`loading`	Loading UI for a segment and its children
`not-found`	Not found UI for a segment and its children
`error`	Error UI for a segment and its children
`global-error`	Global Error UI
`route`	Server-side API endpoint
`template`	Specialized re-rendered Layout UI
`default`	Fallback UI for Parallel Routes

> **Good to know**: `.js`, `.jsx`, or `.tsx` file extensions can be used for special files.

Component Hierarchy

The React components defined in special files of a route segment are rendered in a specific hierarchy:

- `layout.js`
- `template.js`
- `error.js` (React error boundary)
- `loading.js` (React suspense boundary)
- `not-found.js` (React error boundary)
- `page.js` or nested `layout.js`

In a nested route, the components of a segment will be nested **inside** the components of its parent segment.

Colocation

In addition to special files, you have the option to colocate your own files (e.g. components, styles, tests, etc) inside folders in the `app` directory.

This is because while folders define routes, only the contents returned by `page.js` or `route.js` are publicly addressable.

Learn more about Project Organization and Colocation.

Advanced Routing Patterns

The App Router also provides a set of conventions to help you implement more advanced routing patterns. These include:

- Parallel Routes: Allow you to simultaneously show two or more pages in the same view that can be navigated independently. You can use them for split views that have their own sub-navigation. E.g. Dashboards.
- Intercepting Routes: Allow you to intercept a route and show it in the context of another route. You can use these when keeping the context for the current page is important. E.g. Seeing all tasks while editing one task or expanding a photo in a feed.

These patterns allow you to build richer and more complex UIs, democratizing features that were historically complex for small teams and individual developers to implement.

Next Steps

Now that you understand the fundamentals of routing in Next.js, follow the links below to create your first routes:

---

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`
2. On the server, with third-party libraries
3. On the client, via a Route Handler
4. On the client, with third-party libraries.

Fetching Data on the Server with `fetch`

Next.js extends the native `fetch` Web API to allow you to configure the caching and revalidating behavior for each fetch request on the server. React extends `fetch` to automatically memoize fetch requests while rendering a React component tree.

You can use `fetch` with `async`/`await` in Server Components, in Route Handlers, and in Server Actions.

For example:

```
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>
}
```


```
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` and `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 on the server. This means that the data can be fetched at build time or request time, cached, and reused on each data request.

```
// '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 that uses the `POST` method, then it will not be cached.

> **What is the Data Cache?**
>
> The Data Cache is a persistent HTTP cache. Depending on your platform, the cache can scale automatically and be shared across multiple regions.
>
> Learn more about the 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).

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

Alternatively, to revalidate all `fetch` requests in a route segment, you can use the Segment Config Options.

```
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.

On-demand Revalidation

Data can be revalidated on-demand by path (`revalidatePath`) or by cache tag (`revalidateTag`) inside a Server Action or Route Handler.

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`:

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

```
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:

```
'use server'

import { revalidateTag } from 'next/cache'

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

```
'use server'

import { revalidateTag } from 'next/cache'

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

Learn more about 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.

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

View all the available `cache` options in the `fetch` API reference.

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.

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 and React's `cache` function.

Whether the data is cached or not will depend on whether the route segment is statically or dynamically rendered. 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.

Example

In the example below:

- The React `cache` function is used to memoize data requests.
- The `revalidate` option is set to `3600` in the Layout and Page segments, meaning the data will be cached and revalidated at most every hour.

```
import { cache } from 'react'

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

```
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.

```
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)
  // ...
}
```

```
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)
  // ...
}
```

```
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)
  // ...
}
```

```
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 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 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 or TanStack Query. 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.

---

title: Server Actions and Mutations nav_title: Server Actions and Mutations description: Learn how to handle form submissions and data mutations with Next.js. related: description: Learn how to configure Server Actions in Next.js links: - app/api-reference/next-config-js/serverActions

Server Actions are **asynchronous functions** that are executed on the server. They can be used in Server and Client Components to handle form submissions and data mutations in Next.js applications.

> **Watch:** Learn more about forms and mutations with Server Actions → YouTube (10 minutes).

Convention

A Server Action can be defined with the React `"use server"` directive. You can place the directive at the top of an `async` function to mark the function as a Server Action, or at the top of a separate file to mark all exports of that file as Server Actions.

Server Components

Server Components can use the inline function level or module level `"use server"` directive. To inline a Server Action, add `"use server"` to the top of the function body:

```
// Server Component
export default function Page() {
  // Server Action
  async function create() {
    'use server'

    // ...
  }

  return (
    // ...
  )
}
```


```
// Server Component
export default function Page() {
  // Server Action
  async function create() {
    'use server'

    // ...
  }

  return (
    // ...
  )
}
```


Client Components

Client Components can only import actions that use the module-level `"use server"` directive.

To call a Server Action in a Client Component, create a new file and add the `"use server"` directive at the top of it. All functions within the file will be marked as Server Actions that can be reused in both Client and Server Components:

```
'use server'

export async function create() {
  // ...
}
```


```
'use server'

export async function create() {
  // ...
}
```


```
import { create } from '@/app/actions'

export function Button() {
  return (
    // ...
  )
}
```


```
import { create } from '@/app/actions'

export function Button() {
  return (
    // ...
  )
}
```


You can also pass a Server Action to a Client Component as a prop:

```
<ClientComponent updateItem={updateItem} />

'use client'

export default function ClientComponent({ updateItem }) {
  return <form action={updateItem}>{/* ... */}</form>
}
```

Behavior

- Server actions can be invoked using the `action` attribute in a `<form>` element:
  - Server Components support progressive enhancement by default, meaning the form will be submitted even if JavaScript hasn't loaded yet or is disabled.
  - In Client Components, forms invoking Server Actions will queue submissions if JavaScript isn't loaded yet, prioritizing client hydration.
  - After hydration, the browser does not refresh on form submission.
- Server Actions are not limited to `<form>` and can be invoked from event handlers, `useEffect`, third-party libraries, and other form elements like `<button>`.
- Server Actions integrate with the Next.js caching and revalidation architecture. When an action is invoked, Next.js can return both the updated UI and new data in a single server roundtrip.
- Behind the scenes, actions use the `POST` method, and only this HTTP method can invoke them.
- The arguments and return value of Server Actions must be serializable by React. See the React docs for a list of serializable arguments and values.
- Server Actions are functions. This means they can be reused anywhere in your application.
- Server Actions inherit the runtime from the page or layout they are used on.

Examples

Forms

React extends the HTML `<form>` element to allow Server Actions to be invoked with the `action` prop.

When invoked in a form, the action automatically receives the `FormData` object. You don't need to use React `useState` to manage fields, instead, you can extract the data using the native `FormData` methods:

```
export default function Page() {
  async function createInvoice(formData: FormData) {
    'use server'

    const rawFormData = {
      customerId: formData.get('customerId'),
      amount: formData.get('amount'),
      status: formData.get('status'),
    }

    // mutate data
    // revalidate cache
  }

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

```
export default function Page() {
  async function createInvoice(formData) {
    'use server'

    const rawFormData = {
      customerId: formData.get('customerId'),
      amount: formData.get('amount'),
      status: formData.get('status'),
    }

    // mutate data
    // revalidate cache
  }

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

Good to know:

- Example: Form with Loading & Error States
- When working with forms that have many fields, you may want to consider using the `entries()` method with JavaScript's `Object.fromEntries()`. For example: `const rawFormData = Object.fromEntries(formData.entries())`
- See React `<form>` documentation to learn more.

Passing Additional Arguments

You can pass additional arguments to a Server Action using the JavaScript `bind` method.

```
'use client'

import { updateUser } from './actions'

export function UserProfile({ userId }: { userId: string }) {
  const updateUserWithId = updateUser.bind(null, userId)

  return (
    <form action={updateUserWithId}>
      <input type="text" name="name" />
      <button type="submit">Update User Name</button>
    </form>
  )
}
```

```
'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>
  )
}
```

The Server Action will receive the `userId` argument, in addition to the form data:

```
'use server'

export async function updateUser(userId, formData) {
  // ...
}
```

Good to know:

- An alternative is to pass arguments as hidden input fields in the form (e.g. `<input type="hidden" name="userId" value={userId} />`). However, the value will be part of the rendered HTML and will not be encoded.
- `.bind` works in both Server and Client Components. It also supports progressive enhancement.

Pending states

You can use the React `useFormStatus` hook to show a pending state while the form is being submitted.

- `useFormStatus` returns the status for a specific `<form>`, so it **must be defined as a child of the `<form>` element**.

- `useFormStatus` is a React hook and therefore must be used in a Client Component.

```
'use client'

import { useFormStatus } from 'react-dom'

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

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


```
'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 nested in any form:

```
import { SubmitButton } from '@/app/submit-button'
import { createItem } from '@/app/actions'

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


```
import { SubmitButton } from '@/app/submit-button'
import { createItem } from '@/app/actions'

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


Server-side validation and error handling

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

For more advanced server-side validation, you can use a library like zod to validate the form fields before mutating the data:

```
'use server'

import { z } from 'zod'

const schema = z.object({
  email: z.string({
    invalid_type_error: 'Invalid Email',
  }),
})

export default async function createUser(formData: FormData) {
  const validatedFields = schema.safeParse({
    email: formData.get('email'),
  })

  // Return early if the form data is invalid
  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
    }
  }

  // Mutate data
}
```

```
'use server'

import { z } from 'zod'

const schema = z.object({
  email: z.string({
    invalid_type_error: 'Invalid Email',
  }),
})

export default async function createsUser(formData) {
  const validatedFields = schema.safeParse({
    email: formData.get('email'),
  })

  // Return early if the form data is invalid
  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
    }
  }
```

```
  // Mutate data
}
```

Once the fields have been validated on the server, you can return a serializable object in your action and use the React `useFormState` hook to show a message to the user.

- By passing the action to `useFormState`, the action's function signature changes to receive a new `prevState` or `initialState` parameter as its first argument.
- `useFormState` is a React hook and therefore must be used in a Client Component.

```
'use server'

export async function createUser(prevState: any, formData: FormData) {
  // ...
  return {
    message: 'Please enter a valid email',
  }
}
```

```
'use server'

export async function createUser(prevState, formData) {
  // ...
  return {
    message: 'Please enter a valid email',
  }
}
```

Then, you can pass your action to the `useFormState` hook and use the returned `state` to display an error message.

```
'use client'

import { useFormState } from 'react-dom'
import { createUser } from '@/app/actions'

const initialState = {
  message: '',
}

export function Signup() {
  const [state, formAction] = useFormState(createUser, initialState)

  return (
    <form action={formAction}>
      <label htmlFor="email">Email</label>
      <input type="text" id="email" name="email" required />
      {/* ... */}
      <p aria-live="polite" className="sr-only">
        {state?.message}
      </p>
      <button>Sign up</button>
    </form>
  )
}
```

```
'use client'

import { useFormState } from 'react-dom'
import { createUser } from '@/app/actions'

const initialState = {
  message: '',
}

export function Signup() {
  const [state, formAction] = useFormState(createUser, initialState)

  return (
    <form action={formAction}>
      <label htmlFor="email">Email</label>
      <input type="text" id="email" name="email" required />
      {/* ... */}
      <p aria-live="polite" className="sr-only">
        {state?.message}
      </p>
      <button>Sign up</button>
    </form>
  )
}
```

Good to know:

- Before mutating data, you should always ensure a user is also authorized to perform the action. See Authentication and Authorization.

Optimistic updates

You can use the React `useOptimistic` hook to optimistically update the UI before the Server Action finishes, rather than waiting for the response:

```
'use client'

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

type Message = {
  message: string
}

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

```
'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>
  )
}
```

Nested elements

You can invoke a Server Action in elements nested inside `<form>` such as `<button>`, `<input type="submit">`, and `<input type="image">`. These elements accept the `formAction` prop or event handlers.

This is useful in cases where you want to call multiple server actions within a form. For example, you can create a specific `<button>` element for saving a post draft in addition to publishing it. See the React `<form>` docs for more information.

Programmatic form submission

You can trigger a form submission using the `requestSubmit()` method. For example, when the user presses `⌘` + `Enter`, you can listen for the `onKeyDown` event:

```
'use client'

export function Entry() {
  const handleKeyDown = (e: React.KeyboardEvent<HTMLTextAreaElement>) => {
    if (
      (e.ctrlKey || e.metaKey) &&
      (e.key === 'Enter' || e.key === 'NumpadEnter')
    ) {
      e.preventDefault()
      e.currentTarget.form?.requestSubmit()
    }
  }

  return (
    <div>
      <textarea name="entry" rows={20} required onKeyDown={handleKeyDown} />
    </div>
  )
}
```

```
'use client'

export function Entry() {
  const handleKeyDown = (e) => {
    if (
      (e.ctrlKey || e.metaKey) &&
      (e.key === 'Enter' || e.key === 'NumpadEnter')
    ) {
      e.preventDefault()
      e.currentTarget.form?.requestSubmit()
    }
  }

  return (
    <div>
      <textarea name="entry" rows={20} required onKeyDown={handleKeyDown} />
    </div>
  )
}
```

This will trigger the submission of the nearest `<form>` ancestor, which will invoke the Server Action.

Non-form Elements

While it's common to use Server Actions within `<form>` elements, they can also be invoked from other parts of your code such as event handlers and `useEffect`.

Event Handlers

You can invoke a Server Action from event handlers such as `onClick`. For example, to increment a like count:

```
'use server'

export async function incrementLike() {
  // Mutate database
  // Return updated data
}
```

```
'use client'

import { incrementLike } from './actions'
import { useState } from 'react'

export default function LikeButton({ initialLikes }: { initialLikes: number }) {
  const [likes, setLikes] = useState(initialLikes)

  return (
    <>
      <p>Total Likes: {likes}</p>
      <button
        onClick={async () => {
          const updatedLikes = await incrementLike()
          setLikes(updatedLikes)
        }}
      >
        Like
      </button>
    </>
  )
}
```

To improve the user experience, we recommend using other React APIs like `useOptimistic` and `useTransition` to update the UI before the Server Action finishes executing on the server, or to show a pending state.

You can also add event handlers to form elements, for example, to save a form field `onChange`:

```
'use client'

import { publishPost, saveDraft } from './actions'

export default function EditPost() {
  return (
    <form action={publishPost}>
      <textarea
        name="content"
        onChange={async (e) => {
          await saveDraft(e.target.value)
        }}
      />
      <button type="submit">Publish</button>
    </form>
  )
}
```

For cases like this, where multiple events might be fired in quick succession, we recommend **debouncing** to prevent unnecessary Server Action invocations.

`useEffect`

You can use the React `useEffect` hook to invoke a Server Action when the component mounts or a dependency changes. This is useful for mutations that depend on global events or need to be triggered automatically. For example, `onKeyDown` for app shortcuts, an intersection observer hook for infinite scrolling, or when the component mounts to update a view count:

```
'use client'

import { incrementViews } from './actions'
import { useState, useEffect } from 'react'

export default function ViewCount({ initialViews }: { initialViews: number }) {
  const [views, setViews] = useState(initialViews)

  useEffect(() => {
    const updateViews = async () => {
      const updatedViews = await incrementViews()
      setViews(updatedViews)
    }

    updateViews()
  }, [])

  return <p>Total Views: {views}</p>
}
```

Remember to consider the behavior and caveats of `useEffect`.

Error Handling

When an error is thrown, it'll be caught by the nearest `error.js` or `<Suspense>` boundary on the client. We recommend using `try/catch` to return errors to be handled by your UI.

For example, your Server Action might handle errors from creating a new item by returning a message:

```
'use server'

export async function createTodo(prevState: any, formData: FormData) {
  try {
    // Mutate data
  } catch (e) {
```

```
    throw new Error('Failed to create task')
  }
}
```

```
'use server'

export async function createTodo(prevState, formData) {
  try {
    //  Mutate data
  } catch (e) {
    throw new Error('Failed to create task')
  }
}
```

Good to know:

- Aside from throwing the error, you can also return an object to be handled by `useFormState`. See Server-side validation and error handling.

Revalidating data

You can revalidate the Next.js Cache inside your Server Actions with the `revalidatePath` API:

```
'use server'

import { revalidatePath } from 'next/cache'

export async function createPost() {
  try {
    // ...
  } catch (error) {
    // ...
  }

  revalidatePath('/posts')
}
```

```
'use server'

import { revalidatePath } from 'next/cache'

export async function createPost() {
  try {
    // ...
  } catch (error) {
    // ...
  }

  revalidatePath('/posts')
}
```

Or invalidate a specific data fetch with a cache tag using `revalidateTag`:

```
'use server'

import { revalidateTag } from 'next/cache'

export async function createPost() {
  try {
    // ...
  } catch (error) {
    // ...
  }

  revalidateTag('posts')
}
```

```
'use server'

import { revalidateTag } from 'next/cache'

export async function createPost() {
  try {
    // ...
  } catch (error) {
    // ...
  }

  revalidateTag('posts')
}
```

Redirecting

If you would like to redirect the user to a different route after the completion of a Server Action, you can use `redirect` API. `redirect` needs to be called outside of the `try/catch` block:

```
'use server'

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

export async function createPost(id: string) {
  try {
    // ...
  } catch (error) {
    // ...
  }

  revalidateTag('posts') // Update cached posts
  redirect(`/post/${id}`) // Navigate to the new post page
}
```

```
'use server'

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

export async function createPost(id) {
```

```
  try {
    // ...
  } catch (error) {
    // ...
  }

  revalidateTag('posts') // Update cached posts
  redirect(`/post/${id}`) // Navigate to the new post page
}
```

Cookies

You can `get`, `set`, and `delete` cookies inside a Server Action using the `cookies` API:

```
'use server'

import { cookies } from 'next/headers'

export async function exampleAction() {
  // Get cookie
  const value = cookies().get('name')?.value

  // Set cookie
  cookies().set('name', 'Delba')

  // Delete cookie
  cookies().delete('name')
}
```


```
'use server'

import { cookies } from 'next/headers'

export async function exampleAction() {
  // Get cookie
  const value = cookies().get('name')?.value

  // Set cookie
  cookies().set('name', 'Delba')

  // Delete cookie
  cookies().delete('name')
}
```


See additional examples for deleting cookies from Server Actions.

Security

Authentication and authorization

You should treat Server Actions as you would public-facing API endpoints, and ensure that the user is authorized to perform the action. For example:

```
'use server'

import { auth } from './lib'

export function addItem() {
  const { user } = auth()
  if (!user) {
    throw new Error('You must be signed in to perform this action')
  }

  // ...
}
```

Closures and encryption

Defining a Server Action inside a component creates a closure where the action has access to the outer function's scope. For example, the `publish` action has access to the `publishVersion` variable:

```
export default function Page() {
  const publishVersion = await getLatestVersion();

  async function publish(formData: FormData) {
    "use server";
    if (publishVersion !== await getLatestVersion()) {
      throw new Error('The version has changed since pressing publish');
    }
    ...
  }

  return <button action={publish}>Publish</button>;
}
```

```
export default function Page() {
  const publishVersion = await getLatestVersion();

  async function publish() {
    "use server";
    if (publishVersion !== await getLatestVersion()) {
      throw new Error('The version has changed since pressing publish');
    }
    ...
  }

  return <button action={publish}>Publish</button>;
}
```

Closures are useful when you need to capture a *snapshot* of data (e.g. `publishVersion`) at the time of rendering so that it can be used later when the action is invoked.

However, for this to happen, the captured variables are sent to the client and back to the server when the action is invoked. To prevent sensitive data from being exposed to the client, Next.js automatically encrypts the closed-over variables. A new private key is generated for each action every time a Next.js application is built. This means actions can only be invoked for a specific build.

> **Good to know:** We don't recommend relying on encryption alone to prevent sensitive values from being exposed on the client. Instead, you should use the React taint APIs to proactively prevent specific data from being sent to the client.

Overwriting encryption keys (advanced)

When self-hosting your Next.js application across multiple servers, each server instance may end up with a different encryption key, leading to potential inconsistencies.

To mitigate this, you can overwrite the encryption key using the `process.env.NEXT_SERVER_ACTIONS_ENCRYPTION_KEY` environment variable. Specifying this variable ensures that your encryption keys are persistent across builds, and all server instances use the same key.

This is an advanced use case where consistent encryption behavior across multiple deployments is critical for your application. You should consider standard security practices such key rotation and signing.

> **Good to know:** Next.js applications deployed to Vercel automatically handle this.

Allowed origins (advanced)

Since Server Actions can be invoked in a `<form>` element, this opens them up to CSRF attacks.

Behind the scenes, Server Actions use the `POST` method, and only this HTTP method is allowed to invoke them. This prevents most CSRF vulnerabilities in modern browsers, particularly with SameSite cookies being the default.

As an additional protection, Server Actions in Next.js also compare the Origin header to the Host header (or `X-Forwarded-Host`). If these don't match, the request will be aborted. In other words, Server Actions can only be invoked on the same host as the page that hosts it.

For large applications that use reverse proxies or multi-layered backend architectures (where the server API differs from the production domain), it's recommended to use the configuration option `serverActions.allowedOrigins` option to specify a list of safe origins. The option accepts an array of strings.

```
/** @type {import('next').NextConfig} */
module.exports = {
  experimental: {
    serverActions: {
      allowedOrigins: ['my-proxy.com', '*.my-proxy.com'],
    },
  },
}
```

Learn more about Security and Server Actions.

Additional resources

For more information on Server Actions, check out the following React docs:

- `"use server"`
- `<form>`
- `useFormStatus`
- `useFormState`
- `useOptimistic`

---

title: Patterns and Best Practices nav_title: Data Fetching Patterns and Best Practices 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 with Server Components. This allows you to:

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

Then, you can mutate or update data with Server Actions.

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

> **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 are React features that allow you to progressively render and incrementally stream rendered units of the UI to the client.

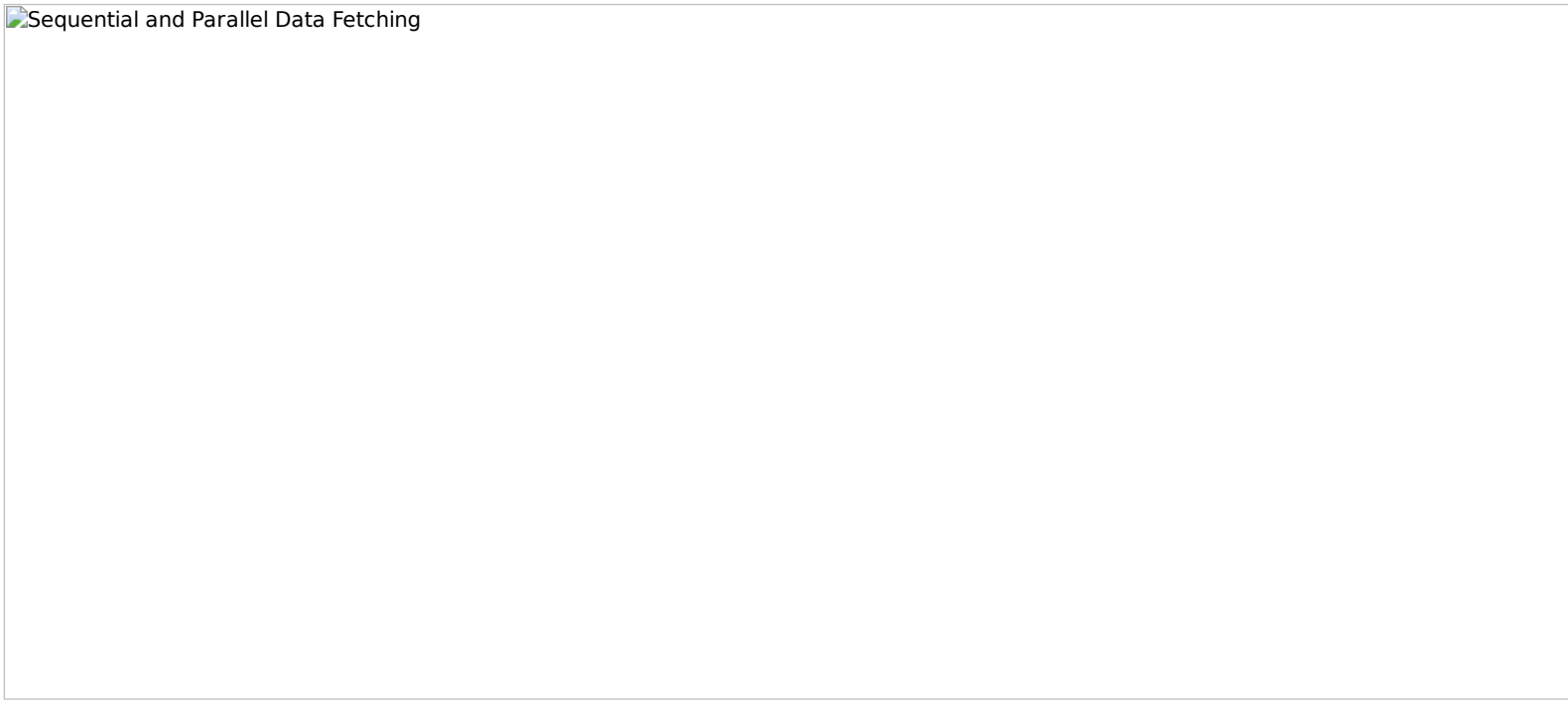
With Server Components and nested layouts, you're able to instantly render parts of the page that do not specifically require data, and show a loading state for parts of the page that are fetching data. This means the user does not have to wait for the entire page to load before they can

start interacting with it.

To learn more about Streaming and Suspense, see the Loading UI and Streaming and Suspense pages.

Parallel and sequential data fetching

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



- 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).

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:

```
// ...

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>
    </>
  )
}
```

```
// ...

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` (for route segments) or React `<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 or the preload pattern.

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:

```
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>
    </>
  )
}

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 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.

```
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)
  // ...
}

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)
  // ...
}

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
}

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.

```
import { cache } from 'react'
import 'server-only'

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

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

```
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.

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 to make sure server data fetching functions are never used on the client.

Preventing sensitive data from being exposed to the client

We recommend using React's taint APIs, `taintObjectReference` and `taintUniqueValue`, to prevent whole object instances or sensitive values from being passed to the client.

To enable tainting in your application, set the Next.js Config `experimental.taint` option to `true`:

```
module.exports = {
  experimental: {
    taint: true,
  },
}
```

Then pass the object or value you want to taint to the `experimental_taintObjectReference` or `experimental_taintUniqueValue` functions:

```
import { queryDataFromDB } from './api'
import {
  experimental_taintObjectReference,
  experimental_taintUniqueValue,
} from 'react'

export async function getUserData() {
  const data = await queryDataFromDB()
  experimental_taintObjectReference(
    'Do not pass the whole user object to the client',
    data
  )
  experimental_taintUniqueValue(
    "Do not pass the user's phone number to the client",
    data,
    data.phoneNumber
  )
  return data
}
```

```
import { queryDataFromDB } from './api'
import {
  experimental_taintObjectReference,
  experimental_taintUniqueValue,
} from 'react'

export async function getUserData() {
  const data = await queryDataFromDB()
  experimental_taintObjectReference(
    'Do not pass the whole user object to the client',
    data
  )
  experimental_taintUniqueValue(
    "Do not pass the user's phone number to the client",
    data,
    data.phoneNumber
  )
  return data
}
```

```
import { getUserData } from './data'

export async function Page() {
  const userData = getUserData()
  return (
    <ClientComponent
      user={userData} // this will cause an error because of taintObjectReference
      phoneNumber={userData.phoneNumber} // this will cause an error because of taintUniqueValue
    />
  )
}
```

```
import { getUserData } from './data'

export async function Page() {
  const userData = getUserData()
  return (
    <ClientComponent
      user={userData} // this will cause an error because of taintObjectReference
```

```
      phoneNumber={userData.phoneNumber} // this will cause an error because of taintUniqueValue
    />
  )
}
```

Learn more about Security and Server Actions.

---

title: Data Fetching description: Learn how to fetch, cache, revalidate, and mutate data with Next.js.

---

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
- Dynamic Rendering
- 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 number 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)**: 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.

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.

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 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. The result is cached and can be pushed to a Content Delivery Network (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 and Data Cache.

Switching to Dynamic Rendering

During rendering, if a dynamic function or uncached data request 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, and you may choose to stream parts of your UI.

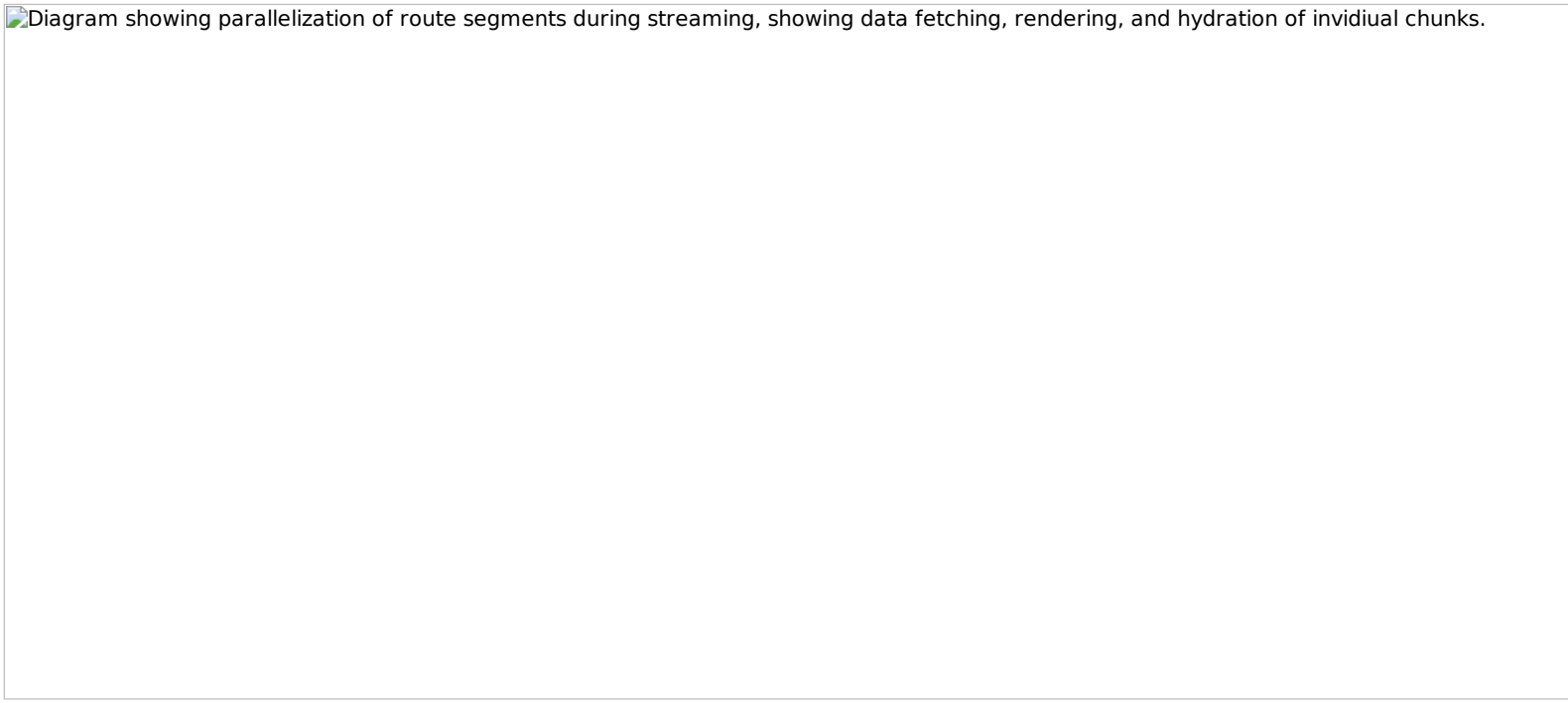
Dynamic Functions

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:

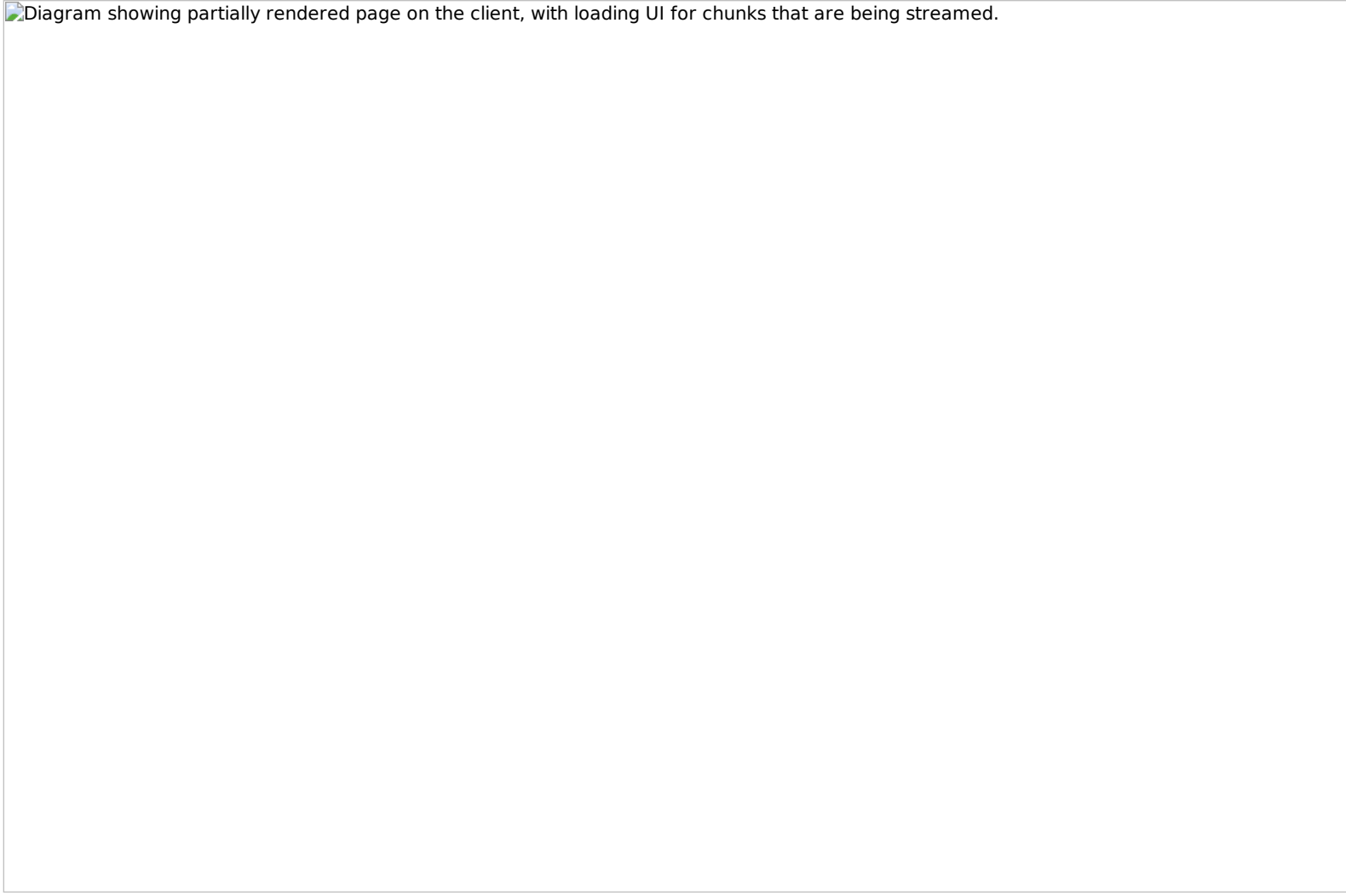
- `cookies()` **and** `headers()`: Using these in a Server Component will opt the whole route into dynamic rendering at request time.
- `searchParams`: Using the Pages prop will opt the page into dynamic rendering at request time.

Using any of these functions will opt the whole route into dynamic rendering at request time.

Streaming



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.



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. See the 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 or 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 at the top of a file, above your imports.

`"use client"` is used to declare a 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.

```
'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>
  )
}
```

```
'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.

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 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` 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. See the 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		
Access backend resources (directly)		
Keep sensitive information on the server (access tokens, API keys, etc)		
Keep large dependencies on the server / Reduce client-side JavaScript		
Add interactivity and event listeners (`onClick()`, `onChange()`, etc)		
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)		
Use browser-only APIs		
Use custom hooks that depend on state, effects, or browser-only APIs		
Use React Class components		

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 (which is not available on the server) or passing data as props, you can use `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 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:

```
export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    headers: {
      authorization: process.env.API_KEY,
    },
  })

  return res.json()
}
```


```
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:

```
npm install server-only
```

Then import the package into any module that contains server-only code:

```
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:

```
'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>
  )
}
```


```
'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:

```
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>
  )
}
```

```
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:

```
'use client'

import { Carousel } from 'acme-carousel'

export default Carousel
```

```
'use client'

import { Carousel } from 'acme-carousel'

export default Carousel
```

Now, you can use `<Carousel />` directly within a Server Component:

```
import Carousel from './carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>

      {/*  Works, since Carousel is a Client Component */}
      <Carousel />
    </div>
  )
}
```

```
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

Context providers are typically rendered near the root of an application to share global concerns, like the current theme. Since React context is not supported in Server Components, trying to create a context at the root of your application will cause an error:

```
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>
  )
}
```

```
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:

```
'use client'

import { createContext } from 'react'

export const ThemeContext = createContext({})

export default function ThemeProvider({
  children,
}: {
  children: React.ReactNode
}) {
  return <ThemeContext.Provider value="dark">{children}</ThemeContext.Provider>
}
```

```
'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:

```
import ThemeProvider from './theme-provider'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <ThemeProvider>{children}</ThemeProvider>
      </body>
    </html>
  )
}
```

```
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, 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 and Vercel Analytics 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.

```
// 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>
    </>
  )
}

// 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 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 or on the server via a Route Handler.

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, 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 and supported pattern 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:

```
'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 />
    </>
  )
}
```

```
'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:

```
'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}
    </>
  )
}
```

```
'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>`:

```
// 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>
  )
}
```

```
// 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.

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	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	Yes	Yes	No
Dynamic Rendering	Yes	Yes	Yes
Data Revalidation w/ `fetch`	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, this limit includes imported packages, fonts and files, and will vary depending on your deployment infrastructure.

Node.js Runtime

Using the Node.js runtime gives you access to all Node.js APIs, and all npm packages that rely on them. However, it's not as fast to start up as routes using the Edge runtime.

Deploying your Next.js application to a Node.js server will require managing, scaling, and configuring your infrastructure. Alternatively, you can consider deploying your Next.js application to a serverless platform like Vercel, which will handle this for you.

Serverless Node.js

Serverless is ideal if you need a scalable solution that can handle more complex computational loads than the Edge Runtime. With Serverless Functions on Vercel, for example, your overall code size is 50MB including imported packages, fonts, and files.

The downside compared to routes using the Edge is that it can take hundreds of milliseconds for Serverless Functions to boot up before they begin processing requests. Depending on the amount of traffic your site receives, this could be a frequent occurrence as the functions are not frequently "warm".

Examples

Segment Runtime Option

You can specify a runtime for individual route segments in your Next.js application. To do so, declare a variable called `runtime` and export it. The variable must be a string, and must have a value of either `'nodejs'` or `'edge'` runtime.

The following example demonstrates a page route segment that exports a `runtime` with a value of `'edge'`:

```
export const runtime = 'edge' // 'nodejs' (default) | 'edge'
```


```
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:

```
export const runtime = 'edge' // 'nodejs' (default) | 'edge'
```


```
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.

> Please refer to the Node.js Docs and Edge Docs for the full list of available APIs. Both runtimes can also support streaming depending on your deployment infrastructure.

---

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 your application code can be executed in: the server and the client.
- The Request-Response Lifecycle that's initiated when a user visits or interacts with your application.
- The 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.

- 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 and Client 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 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 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:

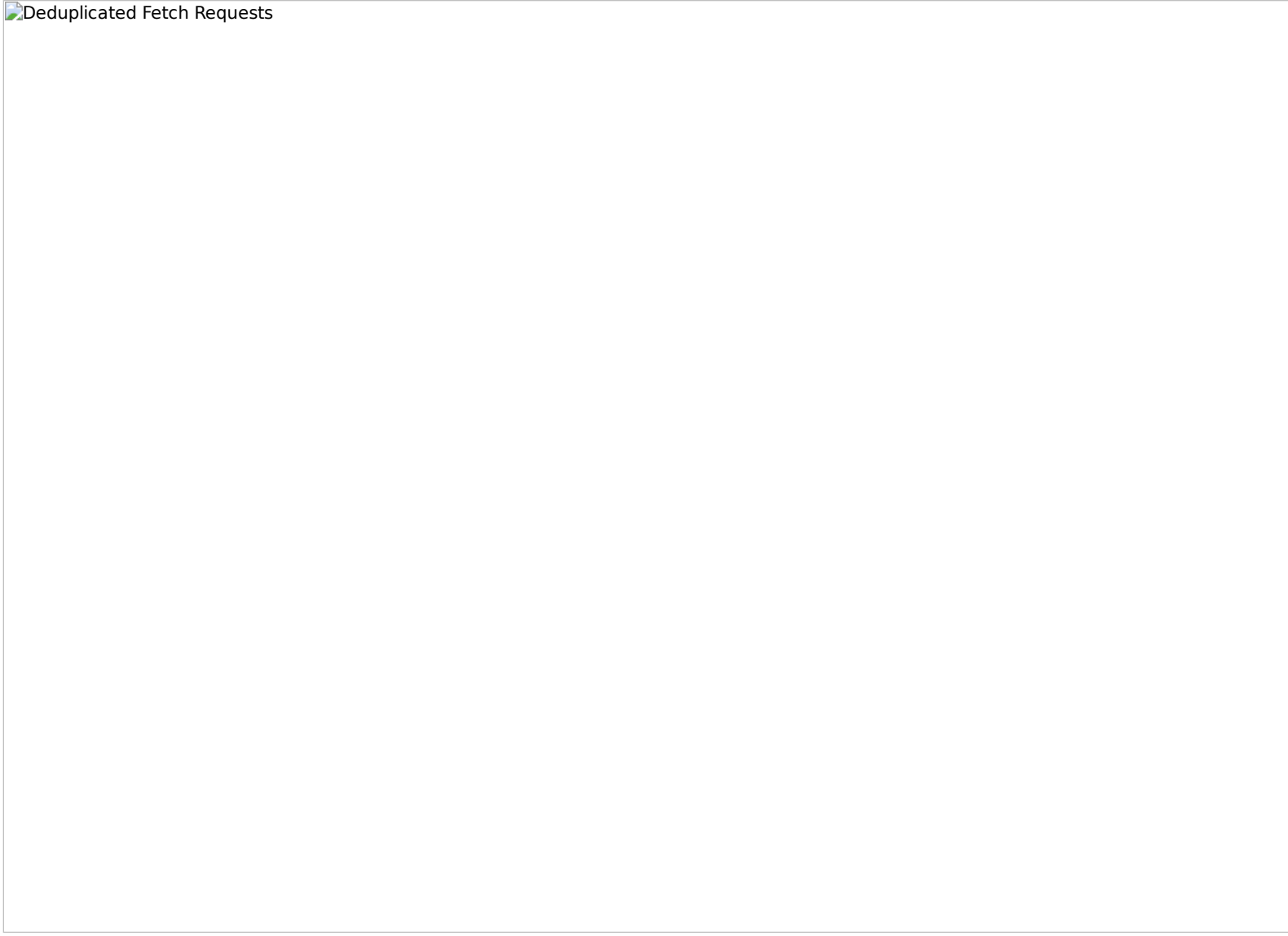
Mechanism	What	Where	Purpose	Duration
Request Memoization	Return values of functions	Server	Re-use data in a React Component tree	Per-request lifecycle
Data Cache	Data	Server	Store data across user requests and deployments	Persistent (can be revalidated)
Full Route Cache	HTML and RSC payload	Server	Reduce rendering cost and improve performance	Persistent (can be revalidated)
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.

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 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.



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.

```
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
```


```
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

- 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] 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.

```
const { signal } = new AbortController()
fetch(url, { signal })
```

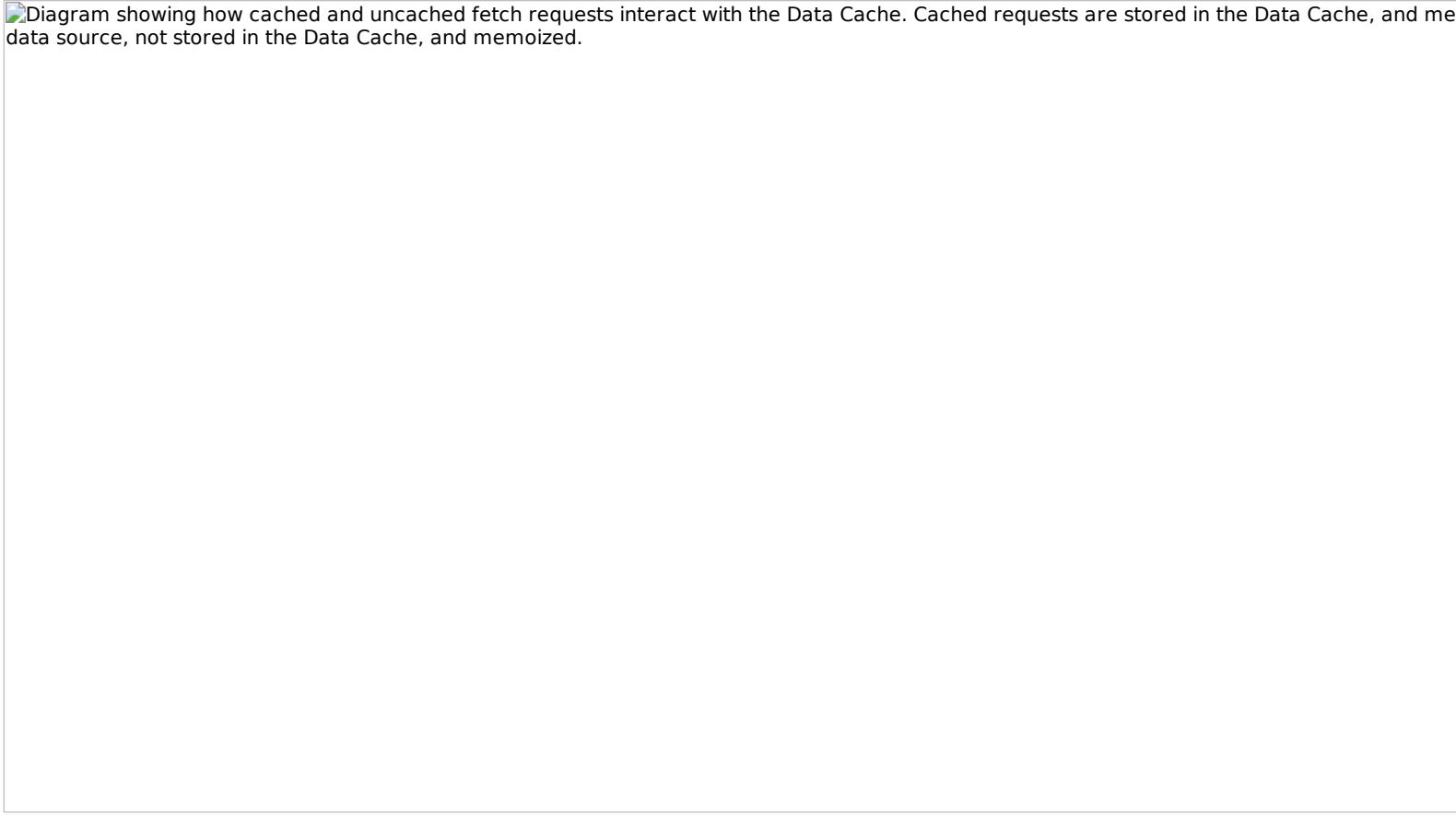
Data Cache

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` and `next.revalidate` options of `fetch` to configure the caching behavior.

How the Data Cache Works



- 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.
- 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).

```
// Revalidate at most every hour
fetch('https://...', { next: { revalidate: 3600 } })
```

Alternatively, you can use Route 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

- 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** behavior.

On-demand Revalidation

Data can be revalidated on-demand by path (`revalidatePath`) or by cache tag (`revalidateTag`).

How On-Demand Revalidation Works

- 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` option to `no-store`. This means data will be fetched whenever `fetch` is called.

```
// Opt out of caching for an individual `fetch` request
fetch(`https://...`, { cache: 'no-store' })
```

Alternatively, you can also use the Route 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.

```
// 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 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 documentation.

2. Next.js Caching on the Server (Full Route Cache)

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 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 - 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:

Learn more about static and dynamic rendering.

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**: Revalidating the 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**: 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**: 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, 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

As a user navigates between routes, Next.js caches visited route segments and prefetches 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.
- 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 or dynamically 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`) or by cache tag with (`revalidateTag`)
  - Using `cookies.set` or `cookies.delete` invalidates the Router Cache to prevent routes that use cookies from becoming stale (e.g. authentication).
- Calling `router.refresh` 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 **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` or `revalidateTag` in a Server Action.

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>`	Cache			
`router.prefetch`	Cache			
`router.refresh`	Revalidate			
`fetch`			Cache	Cache
`fetch options.cache`			Cache or Opt out	
`fetch options.next.revalidate`		Revalidate	Revalidate	
`fetch options.next.tags`		Cache	Cache	
`revalidateTag`	Revalidate (Server Action)	Revalidate	Revalidate	
`revalidatePath`	Revalidate (Server Action)	Revalidate	Revalidate	
`const revalidate`		Revalidate or Opt out	Revalidate or Opt out	
`const dynamic`		Cache or Opt out	Cache or Opt out	
`cookies`	Revalidate (Server Action)	Opt out		
`headers, searchParams`		Opt out		
`generateStaticParams`		Cache		
`React.cache`				Cache
`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.

`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 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 API reference.

`fetch`

Data returned from `fetch` is automatically cached in the Data Cache.

```
// Cached by default. `force-cache` is the default option and can be ommitted.
fetch(`https://...`, { cache: 'force-cache' })
```

See the `fetch` API Reference for more options.

`fetch options.cache`

You can opt out individual `fetch` requests of data caching by setting the `cache` option to `no-store`:

```
// 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 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.

```
// Revalidate at most after 1 hour
fetch(`https://...`, { next: { revalidate: 3600 } })
```

See the `fetch` API reference 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`, 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:

```
// Cache data with a tag
fetch(`https://...`, { next: { tags: ['a', 'b', 'c'] } })
```

Then, call `revalidateTag` with a tag to purge the cache entry:

```
// 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 - 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 - 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.

```
revalidatePath('/')
```

There are two places you can use `revalidatePath`, depending on what you're trying to achieve:

1. Route Handlers - to revalidate data in response to a third party event (e.g. webhook).
2. Server Actions - to revalidate data after a user interaction (e.g. form submission, clicking a button).

See the `revalidatePath` API reference 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

Dynamic functions like `cookies` and `headers`, and the `searchParams` prop in Pages 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` 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 documentation for more options.

generateStaticParams

For dynamic segments (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).

See the `generateStaticParams` API reference.

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.

```
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
})
```

```
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. */}

▼ Examples

- Basic CSS Example

Next.js has built-in support for CSS Modules using the `.module.css` extension.

CSS Modules locally scope CSS by automatically creating a unique class name. This allows you to use the same class name in different files without worrying about collisions. This behavior makes CSS Modules the ideal way to include component-level CSS.

Example

CSS Modules can be imported into any file inside the `app` directory:

```
import styles from './styles.module.css'

export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return <section className={styles.dashboard}>{children}</section>
}
```

```
import styles from './styles.module.css'

export default function DashboardLayout({ children }) {
  return <section className={styles.dashboard}>{children}</section>
}
```

```
.dashboard {
  padding: 24px;
}
```

For example, consider a reusable `Button` component in the `components/` folder:

First, create `components/Button.module.css` with the following content:

```
/*
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:

```
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>
  )
}
```

CSS Modules are an *optional feature* and are **only enabled for files with the `.module.css` extension**. Regular `<link>` stylesheets and global CSS files are still supported.

In production, all CSS Module files will be automatically concatenated into **many minified and code-split** `.css` files. These `.css` files represent hot execution paths in your application, ensuring the minimal amount of CSS is loaded for your application to paint.

Global Styles

Global styles can be imported into any layout, page, or component inside the `app` directory.

> **Good to know**: This is different from the `pages` directory, where you can only import global styles inside the `_app.js` file.

For example, consider a stylesheet named `app/global.css`:

```
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:

```
// 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>
  )
}
```

```
// 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>
  )
}
```

To add a stylesheet to your application, import the CSS file within `pages/_app.js`.

For example, consider the following stylesheet named `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 if not already present. Then, `import` the `styles.css` file.

```
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`**.

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:

```
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>
  )
}
```

External Stylesheets

Stylesheets published by external packages can be imported anywhere in the `app` directory, including colocated components:

```
import 'bootstrap/dist/css/bootstrap.css'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body className="container">{children}</body>
    </html>
  )
}
```

```
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" />`.

Next.js allows you to import CSS files from a JavaScript file. This is possible because Next.js extends the concept of `import` beyond JavaScript.

Import styles from `node_modules`

Since Next.js **9.5.4**, importing a CSS file from `node_modules` is permitted anywhere in your application.

For global stylesheets, like `bootstrap` or `nprogress`, you should import the file inside `pages/_app.js`. For example:

```
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:

```
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>
  )
}
```

Additional Features

Next.js includes additional features to improve the authoring experience of adding styles:

- When running locally with `next dev`, local stylesheets (either global or CSS modules) will take advantage of Fast Refresh to instantly reflect changes as edits are saved.
- When building for production with `next build`, CSS files will be bundled into fewer minified `.css` files to reduce the number of network requests needed to retrieve styles.
- If you disable JavaScript, styles will still be loaded in the production build (`next start`). However, JavaScript is still required for `next dev` to enable Fast Refresh.

---

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. */}

▼ Examples

- With Tailwind CSS

Tailwind CSS is a utility-first CSS framework that works exceptionally well with Next.js.

Installing Tailwind

Install the Tailwind CSS packages and run the `init` command to generate both the `tailwind.config.js` and `postcss.config.js` files:

```
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:

```
/** @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`.

Importing Styles

Add the Tailwind CSS directives that Tailwind will use to inject its generated styles to a Global Stylesheet in your application, for example:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Inside the root layout (`app/layout.tsx`), import the `globals.css` stylesheet to apply the styles to every route in your application.

```
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>
  )
}
```

```
// 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.

```
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

```
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

Importing Styles

Add the Tailwind CSS directives that Tailwind will use to inject its generated styles to a Global Stylesheet in your application, for example:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Inside the custom app file (`pages/_app.js`), import the `globals.css` stylesheet to apply the styles to every route in your application.

```
// 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} />
}
```

```
// 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.

```
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

```
export default function Page() {
  return <h1 className="text-3xl font-bold underline">Hello, Next.js!</h1>
}
```

Usage with Turbopack

As of Next.js 13.1, Tailwind CSS and PostCSS are supported with Turbopack.

---

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. */}

> **Warning:** CSS-in-JS libraries which require runtime JavaScript are not currently supported in Server Components. Using CSS-in-JS with newer React features like Server Components and Streaming requires library authors to support the latest version of React, including concurrent rendering.
>
> We're working with the React team on upstream APIs to handle CSS and JavaScript assets with support for React Server Components and streaming architecture.

The following libraries are supported in Client Components in the `app` directory (alphabetical):

- `chakra-ui`
- `kuma-ui`
- `@mui/material`
- `pandacss`
- `styled-jsx`
- `styled-components`
- `style9`
- `tamagui`
- `tss-react`
- `vanilla-extract`

The following are currently working on support:

- `emotion`

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 or other solutions that output CSS files, like PostCSS or Tailwind CSS.

Configuring CSS-in-JS in `app`

Configuring CSS-in-JS is a three-step opt-in process that involves:

1. A **style registry** to collect all CSS rules in a render.
2. The new `useServerInsertedHTML` hook to inject rules before any content that might use them.
3. A Client Component that wraps your app with the style registry during initial server-side rendering.

`styled-jsx`

Using `styled-jsx` in Client Components requires using `v5.1.0`. First, create a new registry:

```
'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>
}
```

```
'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 with the registry:

```
import StyledJsxRegistry from './registry'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <StyledJsxRegistry>{children}</StyledJsxRegistry>
      </body>
    </html>
  )
}
```

```
import StyledJsxRegistry from './registry'

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <StyledJsxRegistry>{children}</StyledJsxRegistry>
      </body>
    </html>
  )
}
```

View an example here.

Styled Components

Below is an example of how to configure `styled-components@6` or newer:

First, enable styled-components in `next.config.js`.

```
module.exports = {
  compiler: {
    styledComponents: true,
  },
}
```

Then, 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.

```
'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>
  )
}
```

```
'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:

```
import StyledComponentsRegistry from './lib/registry'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <StyledComponentsRegistry>{children}</StyledComponentsRegistry>
      </body>
    </html>
  )
}
```

```
import StyledComponentsRegistry from './lib/registry'

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <StyledComponentsRegistry>{children}</StyledComponentsRegistry>
      </body>
    </html>
  )
}
```

View an example here.

Good to know:

- During server rendering, styles will be extracted to a global registry and flushed to the `<head>` of your HTML. This ensures the style rules are placed before any content that might use them. In the future, we may use an upcoming React feature to determine where to inject the styles.
- During streaming, styles from each chunk will be collected and appended to existing styles. After client-side hydration is complete, `styled-components` will take over as usual and inject any further dynamic styles.
- We specifically use a Client Component at the top level of the tree for the style registry because it's more efficient to extract CSS rules this way. It avoids re-generating styles on subsequent server renders, and prevents them from being sent in the Server Component payload.
- For advanced use cases where you need to configure individual properties of styled-components compilation, you can read our Next.js styled-components API reference to learn more.

▶ Examples

It's possible to use any existing CSS-in-JS solution. The simplest one is inline styles:

```
function HiThere() {
  return <p style={{ color: 'red' }}>hi there</p>
}

export default HiThere
```

We bundle styled-jsx to provide support for isolated scoped CSS. The aim is to support "shadow CSS" similar to Web Components, which unfortunately do not support server-rendering and are JS-only.

See the above examples for other popular CSS-in-JS solutions (like Styled Components).

A component using `styled-jsx` looks like this:

```
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 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.

---

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`:

```
npm install --save-dev sass
```

> **Good to know**:
>
> Sass supports two different syntaxes, each with their own extension. The `.scss` extension requires you use the SCSS syntax, while the `.sass` extension requires you use the Indented Syntax ("Sass").
>
> If you're not sure which to choose, start with the `.scss` extension which is a superset of CSS, and doesn't require you learn the Indented Syntax ("Sass").

Customizing Sass Options

If you want to configure the Sass compiler, use `sassOptions` in `next.config.js`.

```
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:

```
$primary-color: #64ff00;

:export {
  primaryColor: $primary-color;
}
```

```
// maps to root `/` URL

import variables from './variables.module.scss'

export default function Page() {
  return <h1 style={{ color: variables.primaryColor }}>Hello, Next.js!</h1>
}
```

```
import variables from '../styles/variables.module.scss'

export default function MyApp({ Component, pageProps }) {
  return (
    <Layout color={variables.primaryColor}>
      <Component {...pageProps} />
    </Layout>
  )
}
```

---

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. */}

▶ Examples

According to Web Almanac, images account for a huge portion of the typical website's page weight and can have a sizable impact on your website's LCP performance.

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 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).

Usage

```
import Image from 'next/image'
```

You can then define the `src` for your image (either local or remote).

Local Images

To use a local image, `import` your `.jpg`, `.png`, or `.webp` image files.

Next.js will automatically determine the `width` and `height` of your image based on the imported file. These values are used to prevent Cumulative Layout Shift while your image is loading.

```
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
    />
  )
}
```

```
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
    />
  )
}
```

Warning: Dynamic `await import()` or `require()` are *not* supported. The `import` must be static so it can be analyzed at build time.

Remote Images

To use a remote image, the `src` property should be a URL string.

Since Next.js does not have access to remote files during the build process, you'll need to provide the `width`, `height` and optional `blurDataURL` props manually.

The `width` and `height` attributes are used to infer the correct aspect ratio of image and avoid layout shift from the image loading in. The `width` and `height` do *not* determine the rendered size of the image file. Learn more about Image Sizing.

```
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:

```
module.exports = {
  images: {
    remotePatterns: [
```

```
      {
        protocol: 'https',
        hostname: 's3.amazonaws.com',
        port: '',
        pathname: '/my-bucket/**',
      },
    ],
  },
}
```

Learn more about `remotePatterns` configuration. If you want to use relative URLs for the image `src`, use a `loader`.

Domains

Sometimes you may want to optimize a remote image, but still use the built-in Next.js Image Optimization API. To do this, leave the `loader` at its default setting and enter an absolute URL for the Image `src` prop.

To protect your application from malicious users, you must define a list of remote hostnames you intend to use with the `next/image` component.

> Learn more about `remotePatterns` configuration.

Loaders

Note that in the example earlier, a partial URL (`"/me.png"`) is provided for a 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 generation, so that visitors to your site will be served an image that is the right size for their viewport.

The default loader for Next.js applications uses the built-in Image Optimization API, which optimizes images from anywhere on the web, and then serves them directly from the Next.js web server. If you would like to serve your images directly from a CDN or image server, you can write your own loader function with a few lines of JavaScript.

You can define a loader per-image with the `loader` prop, or at the application level with the `loaderFile` configuration.

Priority

You should add the `priority` property to the image that will be the Largest Contentful Paint (LCP) element for each page. Doing so allows Next.js to specially prioritize the image for loading (e.g. through preload tags or priority hints), leading to a meaningful boost in LCP.

The LCP element is typically the largest image or text block visible within the viewport of the page. When you run `next dev`, you'll see a console warning if the LCP element is an `<Image>` without the `priority` property.

Once you've identified the LCP image, you can add the property like this:

```
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>
    </>
  )
}
```

```
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 />
}
```

See more about priority in the `next/image` component documentation.

Image Sizing

One of the ways that images most commonly hurt performance is through *layout shift*, where the image pushes other elements around on the page as it loads in. This performance problem is so annoying to users that it has its own Core Web Vital, called Cumulative Layout Shift. The way to avoid image-based layout shifts is to always size your images. This allows the browser to reserve precisely enough space for the image before it loads.

Because `next/image` is designed to guarantee good performance results, it cannot be used in a way that will contribute to layout shift, and **must** be sized in one of three ways:

1. Automatically, using a static import
2. Explicitly, by including a `width` and `height` property
3. Implicitly, by using fill which causes the image to expand to fill its parent element.

> **What if I don't know the size of my images?**
>
> If you are accessing images from a source without knowledge of the images' sizes, there are several things you can do:
>
> **Use `fill`**
>
> The `fill` prop allows your image to be sized by its parent element. Consider using CSS to give the image's parent element space on the page along `sizes` prop to match any media query break points. You can also use `object-fit` with `fill`, `contain`, or `cover`, and `object-position` to define how the image should occupy that space.

Normalize your images

If you're serving images from a source that you control, consider modifying your image pipeline to normalize the images to a specific size.

Modify your API calls

If your application is retrieving image URLs using an API call (such as to a CMS), you may be able to modify the API call to return the image dimensions along with the URL.

If none of the suggested methods works for sizing your images, the `next/image` component is designed to work well on a page alongside standard `<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:

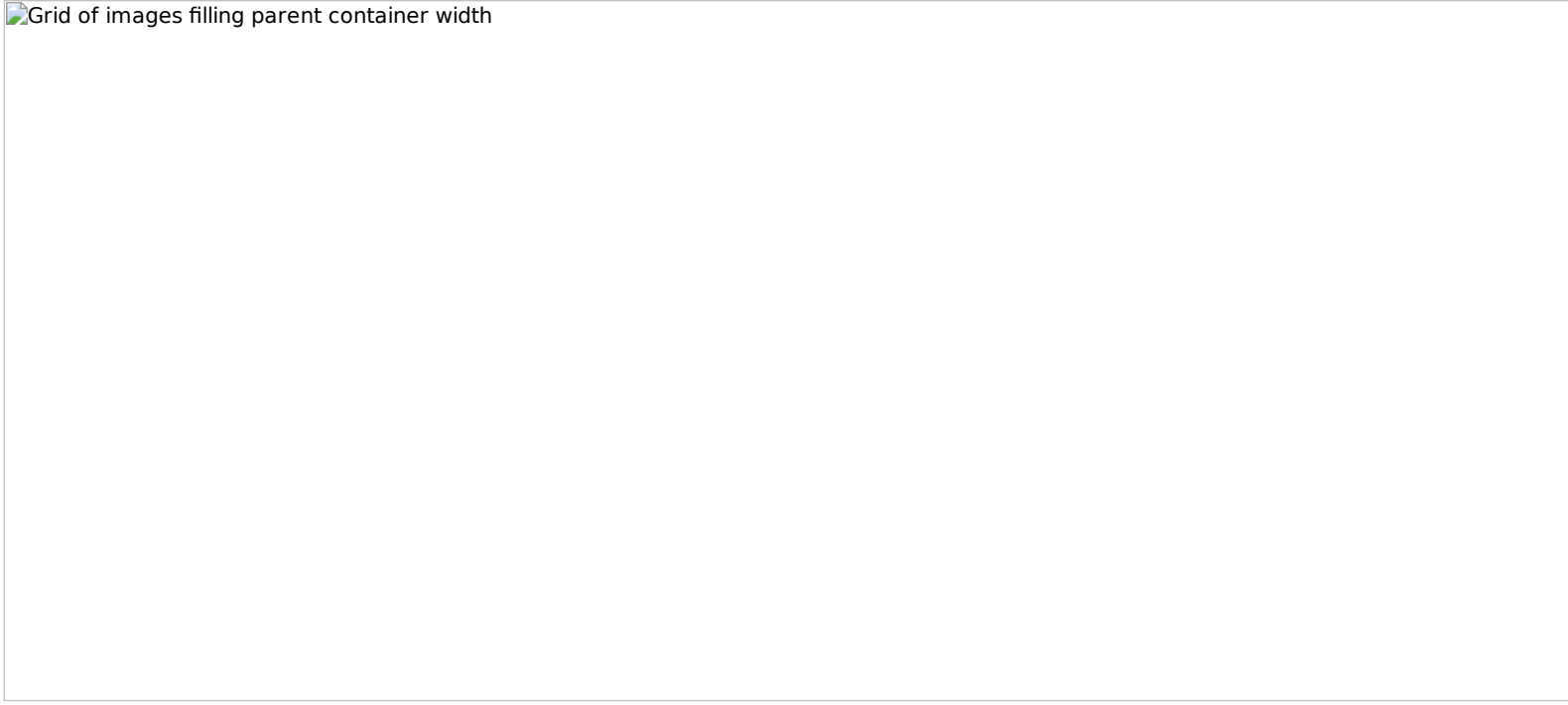
- Use `className` or `style`, not `styled-jsx`.
  - In most cases, we recommend using the `className` prop. This can be an imported CSS Module, a global stylesheet, etc.
  - You can also use the `style` prop to assign inline styles.
  - You cannot use styled-jsx because it's scoped to the current component (unless you mark the style as `global`).
- When using `fill`, the parent element must have `position: relative`
  - This is necessary for the proper rendering of the image element in that layout mode.
- When using `fill`, the parent element must have `display: block`
  - This is the default for `<div>` elements but should be specified otherwise.

Examples

Responsive


```
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




```
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


```
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.

Other Properties

View all properties available to the `next/image` component.

Configuration

The `next/image` component and Next.js Image Optimization API can be configured in the `next.config.js` file. These configurations allow you to enable remote images, define custom image breakpoints, change caching behavior and more.

Read the full image configuration documentation for more information.

---

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` will automatically optimize your fonts (including custom fonts) and remove external network requests for improved privacy and performance.

> **Watch:** Learn more about how to use `next/font` → YouTube (6 minutes).

`next/font` includes **built-in automatic self-hosting** for *any* font file. This means you can optimally load web fonts with zero layout shift, thanks to the underlying CSS `size-adjust` property used.

This new font system also allows you to conveniently use all Google Fonts with performance and privacy in mind. CSS and font files are downloaded at build time and self-hosted with the rest of your static assets. **No requests are sent to Google by the browser.**

Google Fonts

Automatically self-host any Google Font. Fonts are included in the deployment and served from the same domain as your deployment. **No requests are sent to Google by the browser.**

Get started by importing the font you would like to use from `next/font/google` as a function. We recommend using variable fonts for the best performance and flexibility.

```
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>
  )
}
```

```
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**:

```
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>
  )
}

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>
  )
}
```

To use the font in all your pages, add it to `_app.js` file under `/pages` as shown below:

```
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**:

```
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>
  )
}
```

You can specify multiple weights and/or styles by using an array:

```
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`.

Apply the font in `<head>`

You can also use the font without a wrapper and `className` by injecting it inside the `<head>` as follows:

```
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:

```
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>
  )
}
```

Specifying a subset

Google Fonts are automatically subset. This reduces the size of the font file and improves performance. You'll need to define which of these subsets you want to preload. Failing to specify any subsets while `preload` is `true` will result in a warning.

This can be done by adding it to the function call:

```
const inter = Inter({ subsets: ['latin'] })
```

```
const inter = Inter({ subsets: ['latin'] })
```

```
const inter = Inter({ subsets: ['latin'] })
```

View the Font API Reference for more information.

Using Multiple Fonts

You can import and use multiple fonts in your application. There are two approaches you can take.

The first approach is to create a utility function that exports a font, imports it, and applies its `className` where needed. This ensures the font is preloaded only when it's rendered:

```
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',
})
```

```
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',
})
```

```
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>
  )
}
```

```
import { inter } from './fonts'

export default function Layout({ children }) {
  return (
    <html lang="en" className={inter.className}>
      <body>
        <div>{children}</div>
      </body>
    </html>
  )
}
```

```
import { roboto_mono } from './fonts'

export default function Page() {
  return (
    <>
      <h1 className={roboto_mono.className}>My page</h1>
    </>
  )
}
```

```
import { roboto_mono } from './fonts'

export default function Page() {
  return (
    <>
      <h1 className={roboto_mono.className}>My page</h1>
    </>
  )
}
```

In the example above, `Inter` will be applied globally, and `Roboto Mono` can be imported and applied as needed.

Alternatively, you can create a CSS variable and use it with your preferred CSS solution:

```
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>
  )
}

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>
  )
}

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 for the best performance and flexibility.

```
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>
  )
}

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>
  )
}

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>
  )
}
```

If you want to use multiple files for a single font family, `src` can be an array:

```
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 for more information.

With Tailwind CSS

`next/font` can be used with Tailwind CSS through a CSS variable.

In the example below, we use the font `Inter` from `next/font/google` (you can use any font from Google or Local Fonts). Load your font with the `variable` option to define your CSS variable name and assign it to `inter`. Then, use `inter.variable` to add the CSS variable to your HTML document.

```
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>
  )
}
```

```
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>
  )
}
```

```
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>
  )
}
```

Finally, add the CSS variable to your Tailwind CSS config:

```
/** @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

When a font function is called on a page of your site, it is not globally available and preloaded on all routes. Rather, the font is only preloaded on the related routes based on the type of file where it is used:

- If it's a unique page, it is preloaded on the unique route for that page.
- If it's a layout, it is preloaded on all the routes wrapped by the layout.
- If it's the root layout, it is preloaded on all routes.

When a font function is called on a page of your site, it is not globally available and preloaded on all routes. Rather, the font is only preloaded on the related route/s based on the type of file where it is used:

- if it's a unique page, it is preloaded on the unique route for that page
- if it's in the custom App, it is preloaded on all the routes of the site under `/pages`

Reusing fonts

Every time you call the `localFont` or Google font function, that font is hosted as one instance in your application. Therefore, if you load the same font function in multiple files, multiple instances of the same font are hosted. In this situation, it is recommended to do the following:

- Call the font loader function in one shared file
- Export it as a constant
- Import the constant in each file where you would like to use this font

---

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. */}

Layout Scripts

To load a third-party script for multiple routes, import `next/script` and include the script directly in your layout component:

```
import Script from 'next/script'

export default function DashboardLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <>
      <section>{children}</section>
      <Script src="https://example.com/script.js" />
    </>
  )
}
```

```
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.

Application Scripts

To load a third-party script for all routes, import `next/script` and include the script directly in your root layout:

```
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>
  )
}
```

```
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>
  )
}
```

To load a third-party script for all routes, import `next/script` and include the script directly in your custom `_app`:

```
import Script from 'next/script'

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <Component {...pageProps} />
      <Script src="https://example.com/script.js" />
    </>
  )
}
```

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` API reference documentation to learn more about each strategy and their use cases.

Offloading Scripts To A Web Worker (Experimental)

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

Scripts that use the `worker` strategy are offloaded and executed in a web worker with Partytown. This can improve the performance of your site by dedicating the main thread to the rest of your application code.

This strategy is still experimental and can only be used if the `nextScriptWorkers` flag is enabled in `next.config.js`:

```
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:

```
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.

```
import Script from 'next/script'

export default function Home() {
  return (
    <>
      <Script src="https://example.com/script.js" strategy="worker" />
    </>
  )
}
```


```
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 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:

```
<Script id="show-banner">
  {`document.getElementById('banner').classList.remove('hidden')`}
</Script>
```

Or by using the `dangerouslySetInnerHTML` property:

```
<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.

These handlers will only work when `next/script` is imported and used inside of a Client Component where `"use client"` is defined as the first line of code:

```
'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')
        }}
      />
    </>
  )
}
```


```
'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` API reference to learn more about each event handler and view examples.

These handlers will only work when `next/script` is imported and used inside of a Client Component where `"use client"` is defined as the first line of code:


```
import Script from 'next/script'

export default function Page() {
  return (
    <>
      <Script
        src="https://example.com/script.js"
        onLoad={() => {
          console.log('Script has loaded')
        }}
      />
    </>
  )
}
```


```
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` API reference to learn more about each event handler and view examples.

Additional Attributes

There are many DOM attributes that can be assigned to a `<script>` element that are not used by the Script component, like `nonce` or custom data attributes. Including any additional attributes will automatically forward it to the final, optimized `<script>` element that is included in the HTML.

```
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"
      />
    </>
  )
}
```


```
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"
```

```
      />
    </>
  )
}

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"
      />
    </>
  )
}

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"
      />
    </>
  )
}
```

---

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 or a dynamic `generateMetadata` function in a `layout.js` or `page.js` file.
- **File-based Metadata**: Add static or dynamically generated special files to route segments.

With both these options, Next.js will automatically generate the relevant `<head>` elements for your pages. You can also create dynamic OG images using the `ImageResponse` constructor.

Static Metadata

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

```
import type { Metadata } from 'next'

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

export default function Page() {}

export const metadata = {
  title: '...',
  description: '...',
}

export default function Page() {}
```

For all the available options, see the API Reference.

Dynamic Metadata

You can use `generateMetadata` function to `fetch` metadata that requires dynamic values.

```
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) {}

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.

> **Good to know**:
>
> - Both static and dynamic metadata through `generateMetadata` are **only supported in Server Components**.
> - `fetch` requests are automatically memoized for the same data across `generateMetadata`, `generateStaticParams`, Layouts, Pages, and Server Components. React `cache` can be used if `fetch` is unavailable.
> - Next.js will wait for data fetching inside `generateMetadata` to complete before streaming UI to the client. This guarantees the first part of a streamed response includes `<head>` tags.

File-based metadata

These special files are available for metadata:

- favicon.ico, apple-icon.jpg, and icon.jpg
- opengraph-image.jpg and twitter-image.jpg
- robots.txt
- sitemap.xml

You can use these for static metadata, or you can programmatically generate these files with code.

For implementation and examples, see the Metadata Files API Reference and Dynamic Image Generation.

Behavior

File-based metadata has the higher priority and will override any config-based metadata.

Default Fields

There are two default `meta` tags that are always added even if a route doesn't define metadata:

- The meta charset tag sets the character encoding for the website.
- The meta viewport tag sets the viewport width and scale for the website to adjust for different devices.

```
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
```

> **Good to know**: You can overwrite the default `viewport` meta tag.

Ordering

Metadata is evaluated in order, starting from the root segment down to the segment closest to the final `page.js` segment. For example:

1. `app/layout.tsx` (Root Layout)
2. `app/blog/layout.tsx` (Nested Blog Layout)
3. `app/blog/[slug]/page.tsx` (Blog Page)

Merging

Following the evaluation order, Metadata objects exported from multiple segments in the same route are **shallowly** merged together to form the final metadata output of a route. Duplicate keys are **replaced** based on their ordering.

This means metadata with nested fields such as `openGraph` and `robots` that are defined in an earlier segment are **overwritten** by the last segment to define them.

Overwriting fields

```
export const metadata = {
  title: 'Acme',
  openGraph: {
    title: 'Acme',
    description: 'Acme is a...',
  },
}

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:

```
export const openGraphImage = { images: ['http://...'] }

import { openGraphImage } from './shared-metadata'

export const metadata = {
  openGraph: {
    ...openGraphImage,
    title: 'Home',
  },
}

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

```
export const metadata = {
  title: 'Acme',
  openGraph: {
    title: 'Acme',
    description: 'Acme is a...',
  },
}

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, 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`:

```
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 and file-based Metadata. For example, you can use `ImageResponse` in a `opengraph-image.tsx` file to generate Open Graph images at build time or dynamically at request time.

`ImageResponse` supports common CSS properties including flexbox and absolute positioning, custom fonts, text wrapping, centering, and nested images. See the full list of supported CSS properties.

Good to know:

- Examples are available in the Vercel OG Playground.
- `ImageResponse` uses @vercel/og, Satori, and Resvg to convert HTML and CSS into PNG.
- Only the Edge Runtime is supported. The default Node.js runtime will not work.
- Only flexbox and a subset of CSS properties are supported. Advanced layouts (e.g. `display: grid`) will not work.
- Maximum bundle size of `500KB`. The bundle size includes your JSX, CSS, fonts, images, and any other assets. If you exceed the limit, consider reducing the size of any assets or fetching at runtime.
- Only `ttf`, `otf`, and `woff` font formats are supported. To maximize the font parsing speed, `ttf` or `otf` are preferred over `woff`.

JSON-LD

JSON-LD is a format for structured data that can be used by search engines to understand your content. For example, you can use it to describe a person, an event, an organization, a movie, a book, a recipe, and many other types of entities.

Our current recommendation for JSON-LD is to render structured data as a `<script>` tag in your `layout.js` or `page.js` components. For example:

```
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>
  )
}
```

```
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 for Google or the generic Schema Markup Validator.

You can type your JSON-LD with TypeScript using community packages like `schema-dts`:

```
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: Statics Assets in `public` nav_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:

```
import Image from 'next/image'

export function Avatar() {
  return <Image src="/me.png" alt="me" width="64" height="64" />
}
```

Caching

Next.js automatically adds caching headers to immutable assets in the `public` folder. The default caching headers applied are:

```
Cache-Control: public, max-age=31536000, immutable
```

Robots, Favicons, and others

The 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.

For static metadata files, such as `robots.txt`, `favicon.ico`, etc, you should use special metadata files inside the `app` folder.

> 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 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 for persistent file storage.

---

title: Bundle Analyzer description: Analyze the size of your JavaScript bundles using the @next/bundle-analyzer plugin. related: description: Learn more about optimizing your application for production. links: - app/building-your-application/deploying/production-checklist

`@next/bundle-analyzer` is a plugin for Next.js that helps you manage the size of your JavaScript modules. It generates a visual report of the size of each module and their dependencies. You can use the information to remove large dependencies, split your code, or only load some parts when needed, reducing the amount of data transferred to the client.

Installation

Install the plugin by running the following command:

```
npm i @next/bundle-analyzer
# or
yarn add @next/bundle-analyzer
# or
pnpm add @next/bundle-analyzer
```

Then, add the bundle analyzer's settings to your `next.config.js`.

```
const withBundleAnalyzer = require('@next/bundle-analyzer')({
  enabled: process.env.ANALYZE === 'true',
})

/** @type {import('next').NextConfig} */
const nextConfig = {}

module.exports = withBundleAnalyzer(nextConfig)
```

Analyzing your bundles

Run the following command to analyze your bundles:

```
ANALYZE=true npm run build
# or
ANALYZE=true yarn build
# or
ANALYZE=true pnpm build
```

The report will open three new tabs in your browser, which you can inspect. Doing this regularly while you develop and before deploying your site can help you identify large bundles earlier, and architect your application to be more performant.

---

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 in Next.js helps improve the initial loading performance of an application by decreasing the amount of JavaScript needed to render a route.

It allows you to defer loading of **Client Components** and imported libraries, and only include them in the client bundle when they're needed. For example, you might want to defer loading a modal until a user clicks to open it.

There are two ways you can implement lazy loading in Next.js:

1. Using Dynamic Imports with `next/dynamic`
2. Using `React.lazy()` with Suspense

By default, Server Components are automatically code split, and you can use streaming to progressively send pieces of UI from the server to the client. Lazy loading applies to Client Components.

`next/dynamic`

`next/dynamic` is a composite of `React.lazy()` and Suspense. It behaves the same way in the `app` and `pages` directories to allow for incremental migration.

Examples

Importing Client Components

```
'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`:

```
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.

```
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()` 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.

```
'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

```
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()` function:

```
'use client'

export function Hello() {
  return <p>Hello!</p>
}

import dynamic from 'next/dynamic'

const ClientComponent = dynamic(() =>
  import('../components/hello').then((mod) => mod.Hello)
)
```

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.

```
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 returned by `import()`:

```
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`.

```
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.

```
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>
  )
}
```

---

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 to automatically collect and visualize metrics for you.

Build Your Own

```
import { useReportWebVitals } from 'next/web-vitals'

function MyApp({ Component, pageProps }) {
  useReportWebVitals((metric) => {
    console.log(metric)
  })

  return <Component {...pageProps} />
}
```

View the API Reference for more information.

```
'use client'

import { useReportWebVitals } from 'next/web-vitals'

export function WebVitals() {
  useReportWebVitals((metric) => {
    console.log(metric)
  })
}
```

```
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.

View the API Reference for more information.

Web Vitals

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

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

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

```
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} />
}
```

```
'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
      }
      // ...
    }
  })
}
```

```
'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
      }
      // ...
    }
  })
}
```

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:

```
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.

Sending results to external systems

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

```
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, using the `id` value can allow you to construct metric distributions manually (to calculate percentiles, etc.)
>
> ```
> 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.

---

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.

> **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 to add a suffix, you will also need to update the `instrumentation` filename to match.
> - We have created a basic with-opentelemetry example that you can use.

> **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 to add a suffix, you will also need to update the `instrumentation` filename to match.
> - We have created a basic with-opentelemetry example that you can use.

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:

```
import { init } from 'package-init'

export function register() {
  init()
}
```

```
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:

```
export async function register() {
  await import('package-with-side-effect')
}
```

```
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:

```
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation-node')
  }

  if (process.env.NEXT_RUNTIME === 'edge') {
    await import('./instrumentation-edge')
  }
}
```

```
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: 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 for more information about OpenTelemetry and how it works.

This documentation uses terms like *Span*, *Trace* or *Exporter* throughout this doc, all of which can be found in the OpenTelemetry Observability Primer.

Next.js supports OpenTelemetry instrumentation out of the box, which means that we already instrumented Next.js itself. When you enable OpenTelemetry we will automatically wrap all your code like `getStaticProps` in *spans* with helpful attributes.

> **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`:

```
npm install @vercel/otel
```

Next, create a custom `instrumentation.ts` (or `.js`) file in the **root directory** of the project (or inside `src` folder if using one):

Next, create a custom `instrumentation.ts` (or `.js`) file in the **root directory** of the project (or inside `src` folder if using one):

```
import { registerOTel } from '@vercel/otel'

export function register() {
```

```
  registerOTel('next-app')
}
```

```
import { registerOTel } from '@vercel/otel'

export function register() {
  registerOTel('next-app')
}
```

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 to add a suffix, you will also need to update the `instrumentation` filename to match.
- We have created a basic with-opentelemetry example that you can use.

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 to add a suffix, you will also need to update the `instrumentation` filename to match.
- We have created a basic with-opentelemetry example that you can use.

Manual OpenTelemetry configuration

If our wrapper `@vercel/otel` doesn't suit your needs, you can configure OpenTelemetry manually.

Firstly you need to install OpenTelemetry packages:

```
npm install @opentelemetry/sdk-node @opentelemetry/resources @opentelemetry/semantic-conventions @opentelemetry/sdk-trace-node @opentelemetry/exporter-
```

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:

```
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation.node.ts')
  }
}
```

```
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation.node.js')
  }
}
```

```
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()
```

```
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.

If everything works well you should be able to see the root server span labeled as `GET /requested/pathname`. All other spans from that particular trace will be nested under it.

Next.js traces more spans than are emitted by default. To see more spans, you must set `NEXT_OTEL_VERBOSE=1`.

Deployment

Using OpenTelemetry Collector

When you are deploying with OpenTelemetry Collector, you can use `@vercel/otel`. It will work both on Vercel and when self-hosted.

Deploying on Vercel

We made sure that OpenTelemetry works out of the box on Vercel.

Follow Vercel documentation to connect your project to an observability provider.

Self-hosting

Deploying to other platforms is also straightforward. You will need to spin up your own OpenTelemetry Collector to receive and process the telemetry data from your Next.js app.

To do this, follow the OpenTelemetry Collector Getting Started guide, which will walk you through setting up the collector and configuring it to receive data from your Next.js app.

Once you have your collector up and running, you can deploy your Next.js app to your chosen platform following their respective deployment guides.

Custom Exporters

We recommend using OpenTelemetry Collector. If that is not possible on your platform, you can use a custom OpenTelemetry exporter with manual OpenTelemetry configuration

Custom Spans

You can add a custom span with OpenTelemetry APIs.

```
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:

```
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. We also add some custom attributes under the `next` namespace:

- `next.span_name` - duplicates span name
- `next.span_type` - each span type has a unique identifier
- `next.route` - The route pattern of the request (e.g., `/[param]/user`).
- `next.page`
  - This is an internal value used by an app router.
  - You can think about it as a route to a special file (like `page.ts`, `layout.ts`, `loading.ts` and others)
  - It can be used as a unique identifier only when paired with `next.route` because `/layout` can be used to identify both `/(groupA)/layout.ts` and `/(groupB)/layout.ts`

`[http.method] [next.route]`

- `next.span_type`: `BaseServer.handleRequest`

This span represents the root span for each incoming request to your Next.js application. It tracks the HTTP method, route, target, and status code of the request.

Attributes:

- Common HTTP attributes
  - `http.method`
  - `http.status_code`
- Server HTTP attributes
  - `http.route`
  - `http.target`
- `next.span_name`
- `next.span_type`
- `next.route`

`render route (app) [next.route]`

- `next.span_type`: `AppRender.getBodyResult`.

This span represents the process of rendering a route in the app router.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`fetch [http.method] [http.url]`

- `next.span_type`: `AppRender.fetch`

This span represents the fetch request executed in your code.

Attributes:

- Common HTTP attributes
  - `http.method`
- Client HTTP attributes
  - `http.url`
  - `net.peer.name`
  - `net.peer.port` (only if specified)
- `next.span_name`
- `next.span_type`

`executing api route (app) [next.route]`

- `next.span_type`: `AppRouteRouteHandlers.runHandler`.

This span represents the execution of an API route handler in the app router.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`getServerSideProps [next.route]`

- `next.span_type`: `Render.getServerSideProps`.

This span represents the execution of `getServerSideProps` for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`getStaticProps [next.route]`

- `next.span_type`: `Render.getStaticProps`.

This span represents the execution of `getStaticProps` for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`render route (pages) [next.route]`

- `next.span_type`: `Render.renderDocument`.

This span represents the process of rendering the document for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`generateMetadata [next.page]`

- `next.span_type`: `ResolveMetadata.generateMetadata`.

This span represents the process of generating metadata for a specific page (a single route can have multiple of these spans).

Attributes:

- `next.span_name`
- `next.span_type`
- `next.page`

---

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:

```
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 container to your page. By default, it fetches the original inline script after hydration occurs on the page.

To load Google Tag Manager for all routes, include the component directly in your root layout and pass in your GTM container ID:

```
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>
  )
}
```

```
import { GoogleTagManager } from '@next/third-parties/google'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
      <GoogleTagManager gtmId="GTM-XYZ" />
    </html>
  )
}
```

To load Google Tag Manager for all routes, include the component directly in your custom `_app` and pass in your GTM container ID:

```
import { GoogleTagManager } from '@next/third-parties/google'

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <Component {...pageProps} />
      <GoogleTagManager gtmId="GTM-XYZ" />
    </>
  )
}
```

To load Google Tag Manager for a single route, include the component in your page file:

```
import { GoogleTagManager } from '@next/third-parties/google'

export default function Page() {
  return <GoogleTagManager gtmId="GTM-XYZ" />
}
```

```
import { GoogleTagManager } from '@next/third-parties/google'

export default function Page() {
  return <GoogleTagManager gtmId="GTM-XYZ" />
}
```

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.

```
'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>
  )
}
```

```
import { sendGTMEvent } from '@next/third-parties/google'

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGTMEvent({ event: 'buttonClicked', value: 'xyz' })}
      >
        Send Event
      </button>
    </div>
  )
}
```

Refer to the Tag Manager developer 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.

Name	Type	Description
`gtmId`	Required	Your GTM container ID. Usually starts with `GTM-`.
`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 Analytics

The `GoogleAnalytics` component can be used to include Google Analytics 4 to your page via the Google tag (`gtag.js`). By default, it fetches the original scripts after hydration occurs on the page.

> **Recommendation**: If Google Tag Manager is already included in your application, you can configure Google Analytics directly using it, rather than including Google Analytics as a separate component. Refer to the documentation to learn more about the differences between Tag Manager and `gtag.js`.

To load Google Analytics for all routes, include the component directly in your root layout and pass in your measurement ID:

```
import { GoogleAnalytics } from '@next/third-parties/google'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>{children}</body>
      <GoogleAnalytics gaId="G-XYZ" />
    </html>
  )
}
```

```
import { GoogleAnalytics } from '@next/third-parties/google'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>{children}</body>
      <GoogleAnalytics gaId="G-XYZ" />
    </html>
  )
}
```

To load Google Analytics for all routes, include the component directly in your custom `_app` and pass in your measurement ID:

```
import { GoogleAnalytics } from '@next/third-parties/google'

export default function MyApp({ Component, pageProps }) {
  return (
    <>
      <Component {...pageProps} />
      <GoogleAnalytics gaId="G-XYZ" />
    </>
  )
}
```

To load Google Analytics for a single route, include the component in your page file:

```
import { GoogleAnalytics } from '@next/third-parties/google'

export default function Page() {
  return <GoogleAnalytics gaId="G-XYZ" />
}
```

```
import { GoogleAnalytics } from '@next/third-parties/google'

export default function Page() {
  return <GoogleAnalytics gaId="G-XYZ" />
}
```

Sending Events

The `sendGAEvent` function can be used to measure user interactions on your page by sending events using the `dataLayer` object. For this function to work, the `<GoogleAnalytics />` component must be included in either a parent layout, page, or component, or directly in the same file.

```
'use client'

import { sendGAEvent } from '@next/third-parties/google'

export function EventButton() {
  return (
    <div>
      <button
        onClick={() => sendGAEvent({ event: 'buttonClicked', value: 'xyz' })}
      >
        Send Event
      </button>
    </div>
  )
}
```

```
import { sendGAEvent } from '@next/third-parties/google'

export function EventButton() {
  return (
    <div>
      <button
```

```
        onClick={() => sendGAEvent({ event: 'buttonClicked', value: 'xyz' })}
      >
        Send Event
      </button>
    </div>
  )
}
```

Refer to the Google Analytics developer documentation to learn more about event parameters.

Tracking Pageviews

Google Analytics automatically tracks pageviews when the browser history state changes. This means that client-side navigations between Next.js routes will send pageview data without any configuration.

To ensure that client-side navigations are being measured correctly, verify that the *"Enhanced Measurement"* property is enabled in your Admin panel and the *"Page changes based on browser history events"* checkbox is selected.

> **Note**: If you decide to manually send pageview events, make sure to disable the default pageview measurement to avoid having duplicate data. Refer to the Google Analytics developer documentation to learn more.

Options

Options to pass to the `<GoogleAnalytics>` component.

Name	Type	Description
`gaId`	Required	Your measurement ID. Usually starts with `G-`.
`dataLayerName`	Optional	Name of the data layer. Defaults to `dataLayer`.

Google Maps Embed

The `GoogleMapsEmbed` component can be used to add a Google Maps Embed to your page. By default, it uses the `loading` attribute to lazy-load the embed below the fold.

```
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"
    />
  )
}
```

```
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"
    />
  )
}
```

Options

Options to pass to the Google Maps Embed. For a full list of options, read the Google Map Embed docs.

Name	Type	Description
`apiKey`	Required	Your api key.
`mode`	Required	Map mode
`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` under the hood.

```
import { YouTubeEmbed } from '@next/third-parties/google'

export default function Page() {
  return <YouTubeEmbed videoid="ogfYd705cRs" height={400} params="controls=0" />
}
```

```
import { YouTubeEmbed } from '@next/third-parties/google'
```

```
export default function Page() {
  return <YouTubeEmbed videoid="ogfYd705cRs" height={400} params="controls=0" />
}
```

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. Params are passed as a query param string. 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. 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.

The Metadata API in Next.js allows you to modify the `<head>` element of a page. You can configure metadata in two ways:

- **Config-based Metadata**: Export a static `metadata` object or a dynamic `generateMetadata` function in a `layout.js` or `page.js` file.
- **File-based Metadata**: Add static or dynamically generated special files to route segments.

Additionally, you can create dynamic Open Graph Images using JSX and CSS with imageResponse constructor.

The Head Component in Next.js allows you to modify the `<head>` of a page. Learn more in the Head Component documentation.

Static Assets

Next.js `/public` folder can be used to serve static assets like images, fonts, and other files. Files inside `/public` can also be cached by CDN providers so that they are delivered efficiently.

Analytics and Monitoring

For large applications, Next.js integrates with popular analytics and monitoring tools to help you understand how your application is performing. Learn more in the Analytics, OpenTelemetry, and Instrumentation guides.

---

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.

As well as a TypeScript Plugin for your editor.

> **Watch:** Learn about the built-in TypeScript plugin → YouTube (3 minutes)

New Projects

`create-next-app` now ships with TypeScript by default.

```
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.

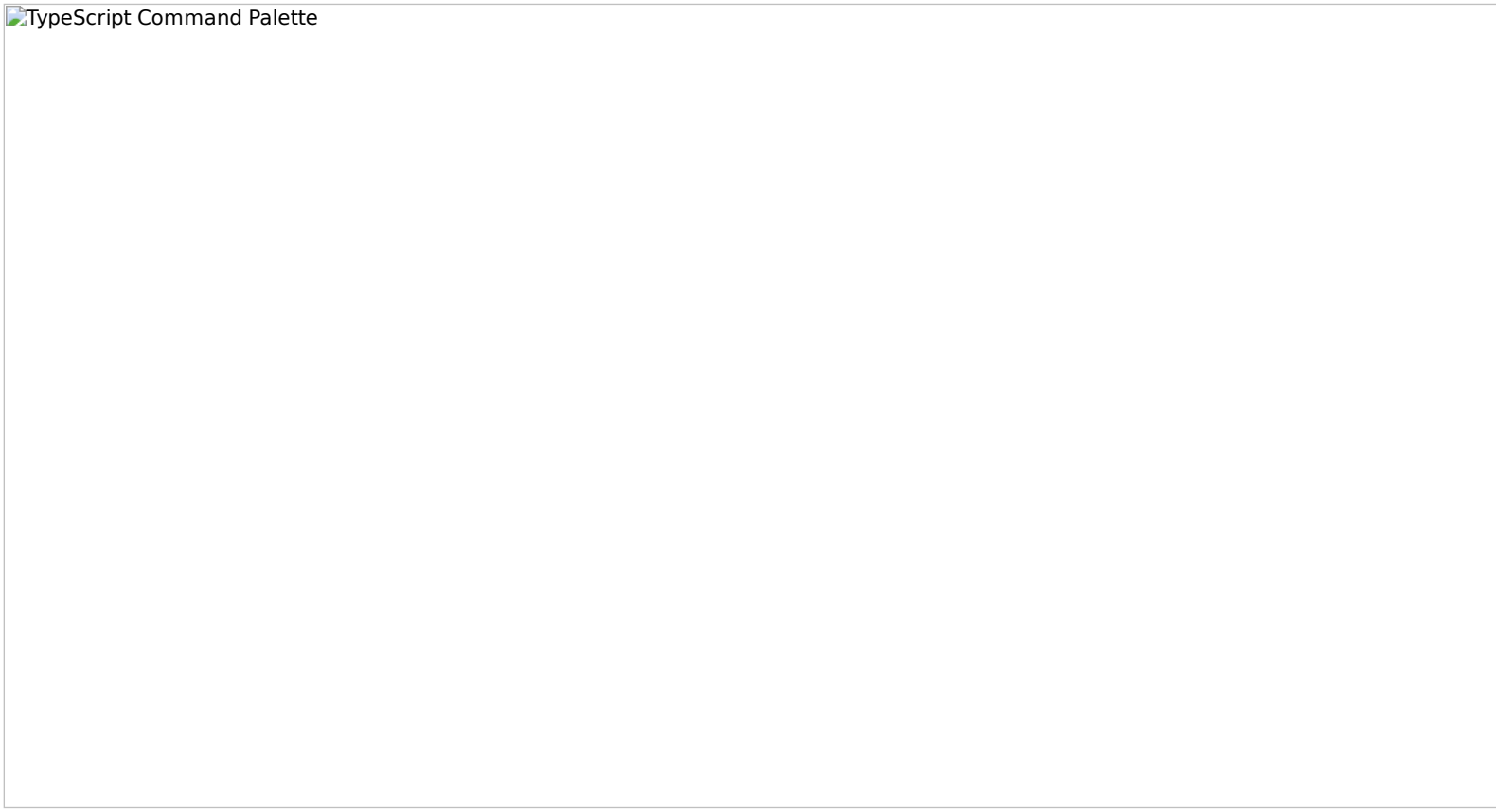
If you already had a `jsconfig.json` file, copy the `paths` compiler option from the old `jsconfig.json` into the new `tsconfig.json` file, and delete the old `jsconfig.json` file.

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"



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 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.

Minimum TypeScript Version

It is highly recommended to be on at least `v4.5.2` of TypeScript to get syntax features such as type modifiers on import names and performance improvements.

Statically Typed Links

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

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

```
/** @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`:

```
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:

```
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 in Next.js 13, this is no longer an issue.

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

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

```
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 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.

Static Generation and Server-side Rendering

For `getStaticProps`, `getStaticPaths`, and `getServerSideProps`, you can use the `GetStaticProps`, `GetStaticPaths`, and `GetServerSideProps` types respectively:

```
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. 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:

```
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:

```
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`, you can use the built-in type `AppProps` and change file name to `./pages/_app.tsx` like so:

```
import type { AppProps } from 'next/app'

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

Path aliases and baseUrl

Next.js automatically supports the `tsconfig.json` `"paths"` and `"baseUrl"` options.

You can learn more about this feature on the Module Path aliases documentation.

You can learn more about this feature on the Module Path aliases documentation.

Type checking next.config.js

The `next.config.js` file must be a JavaScript file as it does not get parsed by Babel or TypeScript, however you can add some type checking in your IDE using JSDoc as below:

```
// @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 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:

```
module.exports = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}
```

Custom Type Declarations

When you need to declare custom types, you might be tempted to modify `next-env.d.ts`. However, this file is automatically generated, so any changes you make will be overwritten. Instead, you should create a new file, let's call it `new-types.d.ts`, and reference it in your `tsconfig.json`:

```
{
  "compilerOptions": {
    "skipLibCheck": true
    //...truncated...
  },
  "include": [
```

```
    "new-types.d.ts",
    "next-env.d.ts",
    ".next/types/**/*.ts",
    "**/*.ts",
    "**/*.tsx"
  ],
  "exclude": ["node_modules"]
}
```

Version Changes

Version	Changes
`v13.2.0`	Statically typed links are available in beta.
`v12.0.0`	SWC is now used by default to compile TypeScript and TSX for faster builds.
`v10.2.1`	Incremental type checking support added when enabled in your `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 experience out of the box. Add `next lint` as a script to `package.json`:

```
{
  "scripts": {
    "lint": "next lint"
  }
}
```

Then run `npm run lint` or `yarn lint`:

```
yarn lint
```

If you don't already have ESLint configured in your application, you will be guided through the installation and configuration process.

```
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. This is the recommended configuration for developers setting up ESLint for the first time.

  ```
  {
    "extends": "next/core-web-vitals"
  }
  ```

- **Base**: Includes Next.js' base ESLint configuration.

  ```
  {
    "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.

> If you do not want ESLint to run during `next build`, refer to the documentation for Ignoring ESLint.

> If you do not want ESLint to run during `next build`, refer to the documentation for Ignoring ESLint.

We recommend using an appropriate integration to view warnings and errors directly in your code editor during development.

ESLint Config

The default configuration (`eslint-config-next`) includes everything you need to have an optimal out-of-the-box linting experience in Next.js. If you do not have ESLint already configured in your application, we recommend using `next lint` to set up ESLint along with this configuration.

> If you would like to use `eslint-config-next` along with other ESLint configurations, refer to the Additional Configurations section to learn how to do so without causing any conflicts.

Recommended rule-sets from the following ESLint plugins are all used within `eslint-config-next`:

- `eslint-plugin-react`
- `eslint-plugin-react-hooks`
- `eslint-plugin-next`

This will take precedence over the configuration from `next.config.js`.

ESLint Plugin

Next.js provides an ESLint plugin, `eslint-plugin-next`, already bundled within the base configuration that makes it possible to catch common issues and problems in a Next.js application. The full set of rules is as follows:

Enabled in the recommended configuration

Rule	Description
@next/next/google-font-display	Enforce font-display behavior with Google Fonts.
@next/next/google-font-preconnect	Ensure `preconnect` is used with Google Fonts.
@next/next/inline-script-id	Enforce `id` attribute on `next/script` components with inline content.
@next/next/next-script-for-ga	Prefer `next/script` component when using the inline script for Google Analytics.
@next/next/no-assign-module-variable	Prevent assignment to the `module` variable.
@next/next/no-async-client-component	Prevent client components from being async functions.
@next/next/no-before-interactive-script-outside-document	Prevent usage of `next/script`'s `beforeInteractive` strategy outside of `pages/_document.js`.
@next/next/no-css-tags	Prevent manual stylesheet tags.
@next/next/no-document-import-in-page	Prevent importing `next/document` outside of `pages/_document.js`.
@next/next/no-duplicate-head	Prevent duplicate usage of `<Head>` in `pages/_document.js`.
@next/next/no-head-element	Prevent usage of `<head>` element.
@next/next/no-head-import-in-document	Prevent usage of `next/head` in `pages/_document.js`.
@next/next/no-html-link-for-pages	Prevent usage of `<a>` elements to navigate to internal Next.js pages.
@next/next/no-img-element	Prevent usage of `<img>` element due to slower LCP and higher bandwidth.
@next/next/no-page-custom-font	Prevent page-only custom fonts.
@next/next/no-script-component-in-head	Prevent usage of `next/script` in `next/head` component.
@next/next/no-styled-jsx-in-document	Prevent usage of `styled-jsx` in `pages/_document.js`.
@next/next/no-sync-scripts	Prevent synchronous scripts.
@next/next/no-title-in-document-head	Prevent usage of `<title>` with `Head` component from `next/document`.
@next/next/no-typos	Prevent common typos in Next.js's data fetching functions
@next/next/no-unwanted-polyfillio	Prevent duplicate polyfills from Polyfill.io.

If you already have ESLint configured in your application, we recommend extending from this plugin directly instead of including `eslint-config-next` unless a few conditions are met. Refer to the Recommended Plugin Ruleset to learn more.

Custom Settings

`rootDir`

If you're using `eslint-plugin-next` in a project where Next.js isn't installed in your root directory (such as a monorepo), you can tell `eslint-plugin-next` where to find your Next.js application using the `settings` property in your `.eslintrc`:

```
{
  "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:

```
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:

```
next lint --dir pages --dir utils --file bar.js
```

Caching

To improve performance, information of files processed by ESLint are cached by default. This is stored in `.next/cache` or in your defined build directory. If you include any ESLint rules that depend on more than the contents of a single source file and need to disable the cache, use the `--no-cache` flag with `next lint`.

To improve performance, information of files processed by ESLint are cached by default. This is stored in `.next/cache` or in your defined build directory. If you include any ESLint rules that depend on more than the contents of a single source file and need to disable the cache, use the `--no-cache` flag with `next lint`.

```
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`:

```
{
  "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.

```
{
  "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.

> The `next/core-web-vitals` entry point is automatically included for new applications built with Create Next App.

Usage With Other Tools

Prettier

ESLint also contains code formatting rules, which can conflict with your existing Prettier setup. We recommend including eslint-config-prettier in your ESLint config to make ESLint and Prettier work together.

First, install the dependency:

```
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:

```
{
  "extends": ["next", "prettier"]
}
```

lint-staged

If you would like to use `next lint` with lint-staged to run the linter on staged git files, you'll have to add the following to the `.lintstagedrc.js` file in the root of your project in order to specify usage of the `--file` flag.

```
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)
- You have `eslint-plugin-import` installed with Node.js and/or TypeScript resolvers defined to handle imports

Then we recommend either removing these settings if you prefer how these properties have been configured within `eslint-config-next` or extending directly from the Next.js ESLint plugin instead:

```
module.exports = {
  extends: [
    //...
    'plugin:@next/next/recommended',
  ],
}
```

The plugin can be installed normally in your project without needing to run `next lint`:

```
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:

```
{
  "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. */}

▶ Examples

Next.js comes with built-in support for environment variables, which allows you to do the following:

- Use `.env.local` to load environment variables
- Bundle environment variables for the browser by prefixing with `NEXT_PUBLIC_`

Loading Environment Variables

Next.js has built-in support for loading environment variables from `.env.local` into `process.env`.

```
DB_HOST=localhost
DB_USER=myuser
DB_PASS=mypassword
```

This loads `process.env.DB_HOST`, `process.env.DB_USER`, and `process.env.DB_PASS` into the Node.js environment automatically allowing you to use them in Next.js data fetching methods and API routes.

For example, using `getStaticProps`:

```
export async function getStaticProps() {
  const db = await myDB.connect({
    host: process.env.DB_HOST,
    username: process.env.DB_USER,
    password: process.env.DB_PASS,
  })
  // ...
}
```

> **Note**: Next.js also supports multiline variables inside of your `.env*` files:
>
> ```
> # .env.local
>
> # you can write with line breaks
> PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
> ...
> Kh9NV...
> ...
> -----END DSA PRIVATE KEY-----"
>
> # or with `
> ` inside double quotes
> PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
> Kh9NV...
> -----END DSA PRIVATE KEY-----
> "
> ```

> **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.

For example:

```
export async function GET() {
  const db = await myDB.connect({
    host: process.env.DB_HOST,
    username: process.env.DB_USER,
    password: process.env.DB_PASS,
  })
  // ...
}
```

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:

```
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:

```
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).

```
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:

```
// 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. 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.

```
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.
- We do not recommend using the runtimeConfig option, as this does not work with the standalone output mode. Instead, we recommend incrementally adopting 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, Environment Variables can be configured in the Project Settings.

All types of Environment Variables should be configured there. Even Environment Variables used in Development – which can be downloaded onto your local device afterwards.

If you've configured Development Environment Variables you can pull them into a `.env.local` for usage on your local machine using the following command:

```
vercel env pull .env.local
```

> **Good to know**: When deploying your Next.js application to Vercel, your environment variables in `.env*` files will not be made available to Edge Runtime, unless their name are prefixed with `NEXT_PUBLIC_`. We strongly recommend managing your environment variables in Project Settings instead, from where all environment variables are available.

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.

```
// 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`
2. `.env.$(NODE_ENV).local`
3. `.env.local` (Not checked when `NODE_ENV` is `test`.)
4. `.env.$(NODE_ENV)`
5. `.env`

For example, if `NODE_ENV` is `development` and you define a variable in both `.env.development.local` and `.env`, the value in `.env.development.local` will be used.

> **Good to know**: The allowed values for `NODE_ENV` are `production`, `development` and `test`.

Good to know

- If you are using a `/src` directory, `.env.*` files should remain in the root of your project.
- 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. */}

▶ Examples

Next.js has in-built support for the `"paths"` and `"baseUrl"` options of `tsconfig.json` and `jsconfig.json` files.

These options allow you to alias project directories to absolute paths, making it easier to import modules. For example:

```
// 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:

```
{
  "compilerOptions": {
    "baseUrl": "."
  }
}
```

```
export default function Button() {
  return <button>Click me</button>
}
```

```
export default function Button() {
  return <button>Click me</button>
}
```

```
import Button from 'components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
```

```
      <Button />
    </>
  )
}
```

```
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/*`:

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/components/*": ["components/*"]
    }
  }
}
```

```
export default function Button() {
  return <button>Click me</button>
}
```

```
export default function Button() {
  return <button>Click me</button>
}
```

```
import Button from '@/components/button'

export default function HomePage() {
  return (
    <>
      <h1>Hello World</h1>
      <Button />
    </>
  )
}
```

```
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:

```
// tsconfig.json or jsconfig.json
{
  "compilerOptions": {
    "baseUrl": "src/",
    "paths": {
      "@/styles/*": ["styles/*"],
      "@/components/*": ["components/*"]
    }
  }
}

// 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 is a lightweight markup language used to format text. It allows you to write using plain text syntax and convert it to structurally valid HTML. It's commonly used for writing content on websites and blogs.

You write...

```
I **love** using [Next.js](https://nextjs.org/)
```

Output:

```
<p>I <strong>love</strong> using <a href="https://nextjs.org/">Next.js</a></p>
```

MDX is a superset of markdown that lets you write JSX directly in your markdown files. It is a powerful way to add dynamic interactivity and embed React components within your content.

Next.js can support both local MDX content inside your application, as well as remote MDX files fetched dynamically on the server. The Next.js plugin handles transforming markdown and React components into HTML, including support for usage in Server Components (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:

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

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.

```
import type { MDXComponents } from 'mdx/types'

export function useMDXComponents(components: MDXComponents): MDXComponents {
  return {
    ...components,
  }
}

export function useMDXComponents(components) {
  return {
    ...components,
  }
}
```

Update the `next.config.js` file at your project's root to configure it to use MDX:

```
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)
```

Then, create a new MDX page within the `/app` directory:

```
  your-project
  ├── app
  │   └── my-mdx-page
  │       └── page.mdx
  └── package.json
```

Then, create a new MDX page within the `/pages` directory:

```
  your-project
  ├── pages
  │   └── my-mdx-page.mdx
  └── package.json
```

Now you can use markdown and import React components directly inside your MDX page:

```
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. A popular community packages for this use is `next-mdx-remote`.

> **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`:

```
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} />
}

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} />
}

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 } }
}

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 } }
}
```

Navigating to the `/my-mdx-page-remote` route should display your rendered MDX.

Layouts

To share a layout amongst MDX pages, you can use the built-in layouts support with the App Router.

```
export default function MdxLayout({ children }: { children: React.ReactNode }) {
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}</div>
}

export default function MdxLayout({ children }) {
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}</div>
}
```

To share a layout around MDX pages, create a layout component:

```
export default function MdxLayout({ children }: { children: React.ReactNode }) {
  // Create any shared layout or styles here
  return <div style={{ color: 'blue' }}>{children}</div>
}

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:

```
import MdxLayout from '../components/mdx-layout'

# Welcome to my MDX page!

export default function MDXPage({ children }) {
  return <MdxLayout>{children}</MdxLayout>;

}
```

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.

```
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
- remark-mdx-frontmatter
- gray-matter.

To access page metadata with `@next/mdx`, you can export a metadata object from within the `.mdx` file:

```
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:

```
This is a list in markdown:

- One
- Two
- Three
```

The above generates the following `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.

To do this, open the `mdx-components.tsx` file at the root of your application and add custom elements:

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:

```
import type { MDXComponents } from 'mdx/types'
import Image, { ImageProps } 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 as ImageProps)}
      />
    ),
    ...components,
  }
}
```

```
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:

```
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, linking headings, generating a table of contents, 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`:

```
module.exports = withMDX({
  experimental: {
    mdxRs: true,
  },
})
```

Helpful Links

- MDX
- `@next/mdx`
- remark
- rehype
- Markdoc

---

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.

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.

---

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 only for this specific case.

Next.js has a feature called **Draft Mode** which solves this problem. Here are instructions on how to use it.

Step 1: Create and access the Route Handler

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

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

```
// 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')
}
```

```
// 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`

```
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`.

```
// 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)
}
```

```
// 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`.

```
// 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>
  )
}

// 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.

```
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()`:

```
import { draftMode } from 'next/headers'

export async function GET(request: Request) {
  draftMode().disable()
  return new Response('Draft mode is disabled')
}

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`, 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) 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.

▶ Examples

Nonces

A 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 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:

```
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
}
```

```
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`.

We recommend ignoring matching prefetches (from `next/link`) and static assets that don't need the CSP header.

```
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' },
      ],
    },
  ],
}
```

```
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 using `headers`:

```
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}
    />
  )
}
```

```
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}
    />
  )
}
```

Without Nonces

For applications that do not require nonces, you can set the CSP header directly in your `next.config.js` file:

```
const cspHeader = `
    default-src 'self';
    script-src 'self' 'unsafe-eval' 'unsafe-inline';
    style-src 'self' 'unsafe-inline';
    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;
`

module.exports = {
  async headers() {
    return [
      {
        source: '/(.*)',
        headers: [
          {
            key: 'Content-Security-Policy',
            value: cspHeader.replace(/
/g, ''),
          },
        ],
      },
    ]
  },
}
```

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: Setting up Vitest with Next.js nav_title: Vitest description: Learn how to set up Vitest with Next.js for Unit Testing.

Vite and React Testing Library are frequently used together for **Unit Testing**. This guide will show you how to setup Vitest with Next.js and write your first tests.

> **Good to know:** Since `async` Server Components are new to the React ecosystem, Vitest currently does not support them. While you can still run **unit tests** for synchronous Server and Client Components, we recommend using an **E2E tests** for `async` components.

Quickstart

You can use `create-next-app` with the Next.js with-vitest example to quickly get started:

```
npx create-next-app@latest --example with-vitest with-vitest-app
```

Manual Setup

To manually set up Vitest, install `vitest` and the following packages as dev dependencies:

```
npm install -D vitest @vitejs/plugin-react jsdom @testing-library/react
# or
yarn add -D vitest @vitejs/plugin-react jsdom @testing-library/react @vitejs/plugin-react
# or
pnpm install -D vitest @vitejs/plugin-react jsdom @testing-library/react
# or
bun add -D vitest @vitejs/plugin-react jsdom @testing-library/react
```

Create a `vitest.config.ts|js` file in the root of your project, and add the following options:

```
import { defineConfig } from 'vitest/config'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  test: {
    environment: 'jsdom',
  },
})
```

```
import { defineConfig } from 'vitest/config'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  test: {
    environment: 'jsdom',
  },
})
```

For more information on configuring Vitest, please refer to the Vitest Cofiguration docs.

Then, add a `test` script to your `package.json`:

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "test": "vitest"
  }
}
```

When you run `npm run test`, Vitest will **watch** for changes in your project by default.

Creating your first Vitest Unit Test

Check that everything is working by creating a test to check if the `<Page />` component successfully renders a heading:

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import Link from 'next/link'

export default function Page() {
  return (
```

```
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import { expect, test } from 'vitest'
import { render, screen } from '@testing-library/react'
import Page from '../app/page'

test('Page', () => {
  render(<Page />)
  expect(screen.getByRole('heading', { level: 1, name: 'Home' })).toBeDefined()
})
```

```
import { expect, test } from 'vitest'
import { render, screen } from '@testing-library/react'
import Page from '../app/page'

test('Page', () => {
  render(<Page />)
  expect(screen.getByRole('heading', { level: 1, name: 'Home' })).toBeDefined()
})
```

> **Good to know**: The example above uses the common `__tests__` convention, but test files can also be colocated inside the `app` router.

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import { expect, test } from 'vitest'
import { render, screen } from '@testing-library/react'
import Page from '../pages/index'

test('Page', () => {
  render(<Page />)
  expect(screen.getByRole('heading', { level: 1, name: 'Home' })).toBeDefined()
})
```

```
import { expect, test } from 'vitest'
import { render, screen } from '@testing-library/react'
import Page from '../pages/index'

test('Page', () => {
  render(<Page />)
  expect(screen.getByRole('heading', { level: 1, name: 'Home' })).toBeDefined()
})
```

Running your tests

Then, run the following command to run your tests:

```
npm run test
# or
yarn test
# or
pnpm test
# or
bun test
```

Additional Resources

You may find these resources helpful:

- Next.js with Vitest example
- Vitest Docs
- React Testing Library Docs

---

title: Setting up Jest with Next.js nav_title: Jest description: Learn how to set up Jest with Next.js for Unit Testing and Snapshot Testing.

Jest and React Testing Library are frequently used together for **Unit Testing** and **Snapshot Testing**. This guide will show you how to set up Jest with Next.js and write your first tests.

> **Good to know:** Since `async` Server Components are new to the React ecosystem, Jest currently does not support them. While you can still run **unit tests** for synchronous Server and Client Components, we recommend using an **E2E tests** for `async` components.

Quickstart

You can use `create-next-app` with the Next.js with-jest example to quickly get started:

```
npx create-next-app@latest --example with-jest with-jest-app
```

Manual setup

Since the release of Next.js 12, Next.js now has built-in configuration for Jest.

To set up Jest, install `jest` and the following packages as dev dependencies:

```
npm install -D jest jest-environment-jsdom @testing-library/react @testing-library/jest-dom
# or
yarn add -D jest jest-environment-jsdom @testing-library/react @testing-library/jest-dom
# or
pnpm install -D jest jest-environment-jsdom @testing-library/react @testing-library/jest-dom
```

Generate a basic Jest configuration file by running the following command:

```
npm init jest@latest
# or
yarn create jest@latest
# or
pnpm create jest@latest
```

This will take you through a series of prompts to setup Jest for your project, including automatically creating a `jest.config.ts|js` file.

Update your config file to use `next/jest`. This transformer has all the necessary configuration options for Jest to work with Next.js:

```
import type { Config } from 'jest'
import nextJest from 'next/jest.js'

const createJestConfig = nextJest({
  // Provide the path to your Next.js app to load next.config.js and .env files in your test environment
  dir: './',
})

// Add any custom config to be passed to Jest
const config: Config = {
  coverageProvider: 'v8',
  testEnvironment: 'jsdom',
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.ts'],
}

// createJestConfig is exported this way to ensure that next/jest can load the Next.js config which is async
export default createJestConfig(config)
```

```
const nextJest = require('next/jest')

/** @type {import('jest').Config} */
const createJestConfig = nextJest({
  // Provide the path to your Next.js app to load next.config.js and .env files in your test environment
  dir: './',
})

// Add any custom config to be passed to Jest
const config = {
  coverageProvider: 'v8',
  testEnvironment: 'jsdom',
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.ts'],
}

// createJestConfig is exported this way to ensure that next/jest can load the Next.js config which is async
module.exports = createJestConfig(config)
```

Under the hood, `next/jest` is automatically configuring Jest for you, including:

- Setting up `transform` using the Next.js Compiler
- Auto mocking stylesheets (`.css`, `.module.css`, and their scss variants), image imports and `next/font`
- Loading `.env` (and all variants) into `process.env`
- Ignoring `node_modules` from test resolving and transforms
- Ignoring `.next` from test resolving
- Loading `next.config.js` for flags that enable SWC transforms

  Good to know: To test environment variables directly, load them manually in a separate setup script or in your `jest.config.ts` file. For more information, please see Test Environment Variables.

Setting up Jest (with Babel)

If you opt out of the Next.js Compiler and use Babel instead, you will need to manually configure Jest and install `babel-jest` and `identity-obj-proxy` in addition to the packages above.

Here are the recommended options to configure Jest for Next.js:

```
module.exports = {
  collectCoverage: true,
  // on node 14.x coverage provider v8 offers good speed and more or less good report
  coverageProvider: 'v8',
  collectCoverageFrom: [
    '**/*.{js,jsx,ts,tsx}',
    '!**/*.d.ts',
    '!**/node_modules/**',
    '!<rootDir>/out/**',
    '!<rootDir>/.next/**',
    '!<rootDir>/*.config.js',
    '!<rootDir>/coverage/**',
  ],
  moduleNameMapper: {
    // Handle CSS imports (with CSS modules)
    // https://jestjs.io/docs/webpack#mocking-css-modules
    '^.+\.module\.(css|sass|scss)$': 'identity-obj-proxy',

    // Handle CSS imports (without CSS modules)
    '^.+\.(css|sass|scss)$': '<rootDir>/__mocks__/styleMock.js',

    // Handle image imports
    // https://jestjs.io/docs/webpack#handling-static-assets
```

```
    '^.+\.(png|jpg|jpeg|gif|webp|avif|ico|bmp|svg)$/i': `<rootDir>/__mocks__/fileMock.js`,

    // Handle module aliases
    '^@/components/(.*)$': '<rootDir>/components/$1',

    // Handle @next/font
    '@next/font/(.*)': `<rootDir>/__mocks__/nextFontMock.js`,
    // Handle next/font
    'next/font/(.*)': `<rootDir>/__mocks__/nextFontMock.js`,
    // Disable server-only
    'server-only': `<rootDir>/__mocks__/empty.js`,
  },
  // Add more setup options before each test is run
  // setupFilesAfterEnv: ['<rootDir>/jest.setup.js'],
  testPathIgnorePatterns: ['<rootDir>/node_modules/', '<rootDir>/.next/'],
  testEnvironment: 'jsdom',
  transform: {
    // Use babel-jest to transpile tests with the next/babel preset
    // https://jestjs.io/docs/configuration#transform-objectstring-pathtotransformer--pathtotransformer-object
    '^.+\.(js|jsx|ts|tsx)$': ['babel-jest', { presets: ['next/babel'] }],
  },
  transformIgnorePatterns: [
    '/node_modules/',
    '^.+\.module\.(css|sass|scss)$',
  ],
}
```

You can learn more about each configuration option in the Jest docs. We also recommend reviewing `next/jest` configuration to see how Next.js configures Jest.

Handling stylesheets and image imports

Stylesheets and images aren't used in the tests but importing them may cause errors, so they will need to be mocked.

Create the mock files referenced in the configuration above - `fileMock.js` and `styleMock.js` - inside a `__mocks__` directory:

```
module.exports = 'test-file-stub'
```

```
module.exports = {}
```

For more information on handling static assets, please refer to the Jest Docs.

Handling Fonts

To handle fonts, create the `nextFontMock.js` file inside the `__mocks__` directory, and add the following configuration:

```
module.exports = new Proxy(
  {},
  {
    get: function getter() {
      return () => ({
        className: 'className',
        variable: 'variable',
        style: { fontFamily: 'fontFamily' },
      })
    },
  }
)
```

Optional: Handling Absolute Imports and Module Path Aliases

If your project is using Module Path Aliases, you will need to configure Jest to resolve the imports by matching the paths option in the `jsconfig.json` file with the `moduleNameMapper` option in the `jest.config.js` file. For example:

```
{
  "compilerOptions": {
    "module": "esnext",
    "moduleResolution": "bundler",
    "baseUrl": "./",
    "paths": {
      "@/components/*": ["components/*"]
    }
  }
}
```

```
moduleNameMapper: {
  // ...
  '^@/components/(.*)$': '<rootDir>/components/$1',
}
```

Optional: Extend Jest with custom matchers

`@testing-library/jest-dom` includes a set of convenient custom matchers such as `.toBeInTheDocument()` making it easier to write tests. You can import the custom matchers for every test by adding the following option to the Jest configuration file:

```
setupFilesAfterEnv: ['<rootDir>/jest.setup.ts']
```

```
setupFilesAfterEnv: ['<rootDir>/jest.setup.js']
```

Then, inside `jest.setup.ts`, add the following import:

```
import '@testing-library/jest-dom'
```

```
import '@testing-library/jest-dom'
```

> **Good to know:** `extend-expect` was removed in `v6.0`, so if you are using `@testing-library/jest-dom` before version 6, you will need to import `@testing-library/jest-dom/extend-expect` instead.

If you need to add more setup options before each test, you can add them to the `jest.setup.js` file above.

Add a test script to `package.json`:

Finally, add a Jest `test` script to your `package.json` file:

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "test": "jest",
    "test:watch": "jest --watch"
  }
}
```

`jest --watch` will re-run tests when a file is changed. For more Jest CLI options, please refer to the Jest Docs.

Creating your first test:

Your project is now ready to run tests. Create a folder called `__tests__` in your project's root directory.

For example, we can add a test to check if the `<Home />` component successfully renders a heading:

```
export default function Home() {
  return <h1>Home</h1>
}
```

```
import '@testing-library/jest-dom'
import { render, screen } from '@testing-library/react'
import Home from '../pages/index'

describe('Home', () => {
  it('renders a heading', () => {
    render(<Home />)

    const heading = screen.getByRole('heading', { level: 1 })

    expect(heading).toBeInTheDocument()
  })
})
```

For example, we can add a test to check if the `<Page />` component successfully renders a heading:

```
import Link from 'next/link'

export default async function Home() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import '@testing-library/jest-dom'
import { render, screen } from '@testing-library/react'
import Page from '../app/page'

describe('Page', () => {
  it('renders a heading', () => {
    render(<Page />)

    const heading = screen.getByRole('heading', { level: 1 })

    expect(heading).toBeInTheDocument()
  })
})
```

Optionally, add a snapshot test to keep track of any unexpected changes in your component:

```
import { render } from '@testing-library/react'
import Home from '../pages/index'

it('renders homepage unchanged', () => {
  const { container } = render(<Home />)
  expect(container).toMatchSnapshot()
})
```

> **Good to know**: Test files should not be included inside the Pages Router because any files inside the Pages Router are considered routes.

```
import { render } from '@testing-library/react'
import Page from '../app/page'

it('renders homepage unchanged', () => {
  const { container } = render(<Page />)
  expect(container).toMatchSnapshot()
})
```

Running your tests

Then, run the following command to run your tests:

```
npm run test
# or
yarn test
# or
pnpm test
```

Additional Resources

For further reading, you may find these resources helpful:

- Next.js with Jest example
- Jest Docs
- React Testing Library Docs

- Testing Playground - use good testing practices to match elements.

---

title: Setting up Playwright with Next.js nav_title: Playwright description: Learn how to set up Playwright with Next.js for End-to-End (E2E) testing.

Playwright is a testing framework that lets you automate Chromium, Firefox, and WebKit with a single API. You can use it to write **End-to-End (E2E)** testing. This guide will show you how to set up Playwright with Next.js and write your first tests.

Quickstart

The fastest way to get started is to use `create-next-app` with the with-playwright example. This will create a Next.js project complete with Playwright configured.

```
npx create-next-app@latest --example with-playwright with-playwright-app
```

Manual setup

To install Playwright, run the following command:

```
npm init playwright
# or
yarn create playwright
# or
pnpm create playwright
```

This will take you through a series of prompts to setup and configure Playwright for your project, including adding a `playwright.config.ts` file. Please refer to the Playwright installation guide for the step-by-step guide.

Creating your first Playwright E2E test

Create two new Next.js pages:

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

```
import Link from 'next/link'

export default function Home() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import Link from 'next/link'

export default function About() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

Then, add a test to verify that your navigation is working correctly:

```
import { test, expect } from '@playwright/test'

test('should navigate to the about page', async ({ page }) => {
  // Start from the index page (the baseURL is set via the webServer in the playwright.config.ts)
  await page.goto('http://localhost:3000/')
  // Find an element with the text 'About' and click on it
  await page.click('text=About')
  // The new URL should be "/about" (baseURL is used there)
  await expect(page).toHaveURL('http://localhost:3000/about')
  // The new page should contain an h1 with "About"
  await expect(page.locator('h1')).toContainText('About')
})
```

> **Good to know**:
>
> You can use `page.goto("/")` instead of `page.goto("http://localhost:3000/")`, if you add `"baseURL": "http://localhost:3000"` to the `playwright.config.ts` configuration file.

Running your Playwright tests

Playwright will simulate a user navigating your application using three browsers: Chromium, Firefox and Webkit, this requires your Next.js server to be running. We recommend running your tests against your production code to more closely resemble how your application will behave.

Run `npm run build` and `npm run start`, then run `npx playwright test` in another terminal window to run the Playwright tests.

> **Good to know**: Alternatively, you can use the `webServer` feature to let Playwright start the development server and wait until it's fully available.

Running Playwright on Continuous Integration (CI)

Playwright will by default run your tests in the headless mode. To install all the Playwright dependencies, run `npx playwright install-deps`.

You can learn more about Playwright and Continuous Integration from these resources:

- Next.js with Playwright example
- Playwright on your CI provider
- Playwright Discord

---

title: Setting up Cypress with Next.js nav_title: Cypress description: Learn how to set up Cypress with Next.js for End-to-End (E2E) and Component Testing.

Cypress is a test runner used for **End-to-End (E2E)** and **Component Testing**. This page will show you how to set up Cypress with Next.js and write your first tests.

> **Warning:**
>
> - For **component testing**, Cypress currently does not support Next.js version 14 and `async` Server Components. These issues are being tracked. For now, component testing works with Next.js version 13, and we recommend E2E testing for `async` Server Components.
> - Cypress currently does not support TypeScript version 5 with `moduleResolution:"bundler"`. This issue is being tracked.

Quickstart

You can use `create-next-app` with the with-cypress example to quickly get started.

```
npx create-next-app@latest --example with-cypress with-cypress-app
```

Manual setup

To manually set up Cypress, install `cypress` as a dev dependency:

```
npm install -D cypress
# or
yarn add -D cypress
# or
pnpm install -D cypress
```

Add the Cypress `open` command to the `package.json` scripts field:

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint",
    "cypress:open": "cypress open"
  }
}
```

Run Cypress for the first time to open the Cypress testing suite:

```
npm run cypress:open
```

You can choose to configure **E2E Testing** and/or **Component Testing**. Selecting any of these options will automatically create a `cypress.config.js` file and a `cypress` folder in your project.

Creating your first Cypress E2E test

Ensure your `cypress.config.js` file has the following configuration:

```
import { defineConfig } from 'cypress'

export default defineConfig({
  e2e: {
    setupNodeEvents(on, config) {},
  },
})
```

```
const { defineConfig } = require('cypress')

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {},
  },
})
```

Then, create two new Next.js files:

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
```

```
    </div>
  )
}
```

```
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

```
import Link from 'next/link'

export default function Home() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```
import Link from 'next/link'

export default function About() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

Add a test to check your navigation is working correctly:

```
describe('Navigation', () => {
  it('should navigate to the about page', () => {
    // Start from the index page
    cy.visit('http://localhost:3000/')

    // Find a link with an href attribute containing "about" and click it
    cy.get('a[href*="about"]').click()

    // The new url should include "/about"
    cy.url().should('include', '/about')

    // The new page should contain an h1 with "About"
    cy.get('h1').contains('About')
  })
})
```

Running E2E Tests

Cypress will simulate a user navigating your application, this requires your Next.js server to be running. We recommend running your tests against your production code to more closely resemble how your application will behave.

Run `npm run build && npm run start` to build your Next.js application, then run `npm run cypress:open` in another terminal window to start Cypress and run your E2E testing suite.

Good to know:

- You can use `cy.visit("/")` instead of `cy.visit("http://localhost:3000/")` by adding `baseUrl: 'http://localhost:3000'` to the `cypress.config.js` configuration file.
- Alternatively, you can install the `start-server-and-test` package to run the Next.js production server in conjuction with Cypress. After installation, add `"test": "start-server-and-test start http://localhost:3000 cypress"` to your `package.json` scripts field. Remember to rebuild your application after new changes.

Creating your first Cypress component test

Component tests build and mount a specific component without having to bundle your whole application or start a server.

Select **Component Testing** in the Cypress app, then select **Next.js** as your front-end framework. A `cypress/component` folder will be created in your project, and a `cypress.config.js` file will be updated to enable component testing.

Ensure your `cypress.config.js` file has the following configuration:

```
import { defineConfig } from 'cypress'

export default defineConfig({
  component: {
    devServer: {
      framework: 'next',
      bundler: 'webpack',
    },
  },
})
```

```
const { defineConfig } = require('cypress')

module.exports = defineConfig({
  component: {
    devServer: {
      framework: 'next',
      bundler: 'webpack',
    },
  },
})
```

Assuming the same components from the previous section, add a test to validate a component is rendering the expected output:

```
import Page from '../../app/page'

describe('<Page />', () => {
  it('should render and display expected content', () => {
    // Mount the React component for the Home page
    cy.mount(<Page />)

    // The new page should contain an h1 with "Home"
    cy.get('h1').contains('Home')

    // Validate that a link with the expected URL is present
    // Following the link is better suited to an E2E test
    cy.get('a[href="/about"]').should('be.visible')
  })
})

import AboutPage from '../../pages/about'

describe('<AboutPage />', () => {
  it('should render and display expected content', () => {
    // Mount the React component for the About page
    cy.mount(<AboutPage />)

    // The new page should contain an h1 with "About page"
    cy.get('h1').contains('About')

    // Validate that a link with the expected URL is present
    // *Following* the link is better suited to an E2E test
    cy.get('a[href="/"]').should('be.visible')
  })
})
```

Good to know:

- Cypress currently doesn't support component testing for `async` Server Components. We recommend using E2E testing.
- Since component tests do not require a Next.js server, features like `<Image />` that rely on a server being available may not function out-of-the-box.

Running Component Tests

Run `npm run cypress:open` in your terminal to start Cypress and run your component testing suite.

Continuous Integration (CI)

In addition to interactive testing, you can also run Cypress headlessly using the `cypress run` command, which is better suited for CI environments:

```
{
  "scripts": {
    //...
    "e2e": "start-server-and-test dev http://localhost:3000 \"cypress open --e2e\"",
    "e2e:headless": "start-server-and-test dev http://localhost:3000 \"cypress run --e2e\"",
    "component": "cypress open --component",
    "component:headless": "cypress run --component"
  }
}
```

You can learn more about Cypress and Continuous Integration from these resources:

- Next.js with Cypress example
- Cypress Continuous Integration Docs
- Cypress GitHub Actions Guide
- Official Cypress GitHub Action
- Cypress Discord

---

title: Testing description: Learn how to set up Next.js with three commonly used testing tools — Cypress, Playwright, Vitest, and Jest.

In React and Next.js, there are a few different types of tests you can write, each with its own purpose and use cases. This page provides an overview of types and commonly used tools you can use to test your application.

Types of tests

- **Unit testing** involves testing individual units (or blocks of code) in isolation. In React, a unit can be a single function, hook, or component.
  - **Component testing** is a more focused version of unit testing where the primary subject of the tests is React components. This may involve testing how components are rendered, their interaction with props, and their behavior in response to user events.
  - **Integration testing** involves testing how multiple units work together. This can be a combination of components, hooks, and functions.
- **End-to-End (E2E) Testing** involves testing user flows in an environment that simulates real user scenarios, like the browser. This means testing specific tasks (e.g. signup flow) in a production-like environment.
- **Snapshot testing** involves capturing the rendered output of a component and saving it to a snapshot file. When tests run, the current rendered output of the component is compared against the saved snapshot. Changes in the snapshot are used to indicate unexpected changes in behavior.

Async Server Components

Since `async` Server Components are new to the React ecosystem, some tools do not fully support them. In the meantime, we recommend using **End-to-End Testing** over **Unit Testing** for `async` components.

Guides

See the guides below to learn how to set up Next.js with these commonly used testing tools:

---

title: Production Checklist description: Recommendations to ensure the best performance and user experience before taking your Next.js application to production.

Before taking your Next.js application to production, there are some optimizations and patterns you should consider implementing for the best user experience, performance, and security.

This page provides best practices that you can use as a reference when building your application, before going to production, and after deployment - as well as the automatic Next.js optimizations you should be aware of.

Automatic optimizations

These Next.js optimizations are enabled by default and require no configuration:

- **Server Components:** Next.js uses Server Components by default. Server Components run on the server, and don't require JavaScript to render on the client. As such, they have no impact on the size of your client-side JavaScript bundles. You can then use Client Components as needed for interactivity.
- **Code-splitting:** Server Components enable automatic code-splitting by route segments. You may also consider lazy loading Client Components and third-party libraries, where appropriate.
- **Prefetching:** When a link to a new route enters the user's viewport, Next.js prefetches the route in background. This makes navigation to new routes almost instant. You can opt out of prefetching, where appropriate.
- **Static Rendering:** Next.js statically renders Server and Client Components on the server at build time and caches the rendered result to improve your application's performance. You can opt into Dynamic Rendering for specific routes, where appropriate. {/* TODO: Update when PPR is stable */}
- **Caching:** Next.js caches data requests, the rendered result of Server and Client Components, static assets, and more, to reduce the number of network requests to your server, database, and backend services. You may opt out of caching, where appropriate.

- **Code-splitting:** Next.js automatically code-splits your application code by pages. This means only the code needed for the current page is loaded on navigation. You may also consider lazy loading third-party libraries, where appropriate.
- **Prefetching:** When a link to a new route enters the user's viewport, Next.js prefetches the route in background. This makes navigation to new routes almost instant. You can opt out of prefetching, where appropriate.
- **Automatic Static Optmization:** Next.js automatically determines that a page is static (can be pre-rendered) if it has no blocking data requirements. Optimized pages can be cached, and served to the end-user from multiple CDN locations. You may opt into Server-side Rendering, where appropriate.

These defaults aim to improve your application's performance, and reduce the cost and amount of data transferred on each network request.

During development

While building your application, we recommend using the following features to ensure the best performance and user experience:

Routing and rendering

- **Layouts:** Use layouts to share UI across pages and enable partial rendering on navigation.
- **`<Link>` component:** Use the `<Link>` component for client-side navigation and prefetching.
- **Error Handling:** Gracefully handle catch-all errors and 404 errors in production by creating custom error pages.
- **Composition Patterns:** Follow the recommended composition patterns for Server and Client Components, and check the placement of your `"use client"` boundaries to avoid unnecessarily increasing your client-side JavaScript bundle.
- **Dynamic Functions:** Be aware that dynamic functions like `cookies()` and the `searchParams` prop will opt the entire route into Dynamic Rendering (or your whole application if used in the Root Layout). Ensure dynamic function usage is intentional and wrap them in `<Suspense>` boundaries where appropriate.

  Good to know: Partial Prerendering (Experimental) will allow parts of a route to be dynamic without opting the whole route into dynamic rendering.

- **`<Link>` component:** Use the `<Link>` component for client-side navigation and prefetching.
- **Custom Errors:** Gracefully handle 500 and 404 errors

Data fetching and caching

- **Server Components:** Leverage the benefits of fetching data on the server using Server Components.
- **Route Handlers:** Use Route Handlers to access your backend resources from Client Components. But do not call Route Handlers from Server Components to avoid an additional server request.
- **Streaming:** Use Loading UI and React Suspense to progressively send UI from the server to the client, and prevent the whole route from blocking while data is being fetched.
- **Parallel Data Fetching:** Reduce network waterfalls by fetching data in parallel, where appropriate. Also, consider preloading data where appropriate.
- **Data Caching:** Verify whether your data requests are being cached or not, and opt into caching, where appropriate. Ensure requests that don't use `fetch` are cached.
- **Static Images:** Use the `public` directory to automatically cache your application's static assets, e.g. images.

- **API Routes:** Use Route Handlers to access your backend resources, and prevent sensitive secrets from being exposed to the client.
- **Data Caching:** Verify whether your data requests are being cached or not, and opt into caching, where appropriate. Ensure requests that don't use `getStaticProps` are cached where appropriate.
- **Incremental Static Regeneration:** Use Incremental Static Regeneration to update static pages after they've been built, without rebuilding your entire site.
- **Static Images:** Use the `public` directory to automatically cache your application's static assets, e.g. images.

UI and accessibility

- **Forms and Validation:** Use Server Actions to handle form submissions, server-side validation, and handle errors.

- **Font Module:** Optimize fonts by using the Font Module, which automatically hosts your font files with other static assets, removes external network requests, and reduces layout shift.
- **`<Image>` Component:** Optimize images by using the Image Component, which automatically optimizes images, prevents layout shift, and serves them in modern formats like WebP or AVIF.
- **`<Script>` Component:** Optimize third-party scripts by using the Script Component, which automatically defers scripts and prevents them from blocking the main thread.

- **ESLint:** Use the built-in `eslint-plugin-jsx-a11y` plugin to catch accessibility issues early.

Security

- **Tainting:** Prevent sensitive data from being exposed to the client by tainting data objects and/or specific values.
- **Server Actions:** Ensure users are authorized to call Server Actions. Review the the recommended security practices.

- **Environment Variables:** Ensure your `.env.*` files are added to `.gitignore` and only public variables are prefixed with `NEXT_PUBLIC_`.
- **Content Security Policy:** Consider adding a Content Security Policy to protect your application against various security threats such as cross-site scripting, clickjacking, and other code injection attacks.

Metadata and SEO

- **Metadata API:** Use the Metadata API to improve your application's Search Engine Optimization (SEO) by adding page titles, descriptions, and more.
- **Open Graph (OG) images:** Create OG images to prepare your application for social sharing.
- **Sitemaps and Robots:** Help Search Engines crawl and index your pages by generating sitemaps and robots files.

- **`<Head>` Component:** Use the `next/head` component to add page titles, descriptions, and more.

Type safety

- **TypeScript and TS Plugin:** Use TypeScript and the TypeScript plugin for better type-safety, and to help you catch errors early.

Before going to production

Before going to production, you can run `next build` to build your application locally and catch any build errors, then run `next start` to measure the performance of your application in a production-like environment.

Core Web Vitals

- **Lighthouse:** Run lighthouse in incognito to gain a better understanding of how your users will experience your site, and to identify areas for improvement. This is a simulated test and should be paired with looking at field data (such as Core Web Vitals).

- **`useReportWebVitals` hook:** Use this hook to send Core Web Vitals data to analytics tools.

Analyzing bundles

Use the `@next/bundle-analyzer` plugin to analyze the size of your JavaScript bundles and identify large modules and dependencies that might be impacting your application's performance.

Additionally, the following tools can you understand the impact of adding new dependencies to your application:

- Import Cost
- Package Phobia
- Bundle Phobia
- bundlejs

After deployment

Depending on where you deploy your application, you might have access to additional tools and integrations to help you monitor and improve your application's performance.

For Vercel deployments, we recommend the following:

- **Analytics:** A built-in analytics dashboard to help you understand your application's traffic, including the number of unique visitors, page views, and more.
- **Speed Insights:** Real-world performance insights based on visitor data, offering a practical view of how your website is performing in the field.
- **Logging:** Runtime and Activity logs to help you debug issues and monitor your application in production. Alternatively, see the integrations page for a list of third-party tools and services.

  Good to know:

  To get a comprehensive understanding of the best practices for production deployments on Vercel, including detailed strategies for improving website performance, refer to the Vercel Production Checklist.

Following these recommendations will help you build a faster, more reliable, and secure application for your users.

---

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.

> **Good to know**: We recommend using the App Router for enhanced static export support.

Configuration

To enable a static export, change the output mode inside `next.config.js`:

```
/**
 * @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.

You can utilize `getStaticProps` and `getStaticPaths` to generate an HTML file for each page in your `pages` directory (or more for dynamic routes).

Supported Features

The 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.

```
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 to memoize requests.

```
'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
}
```

```
'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:

```
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>
    </>
  )
}
```

```
import Link from 'next/link'

export default function Page() {
  return (
    <>
      <h1>Index Page</h1>
      <p>
        <Link href="/other">Other Page</Link>
      </p>
    </>
  )
}
```

Supported Features

The majority of core Next.js features needed to build a static site are supported, including:

- Dynamic Routes when using `getStaticPaths`
- Prefetching with `next/link`
- Preloading JavaScript
- Dynamic Imports
- Any styling options (e.g. CSS Modules, styled-jsx)
- Client-side data fetching
- `getStaticProps`
- `getStaticPaths`

Image Optimization

Image Optimization through `next/image` can be used with a static export by defining a custom image loader in `next.config.js`. For example, you can optimize images with a service like Cloudinary:

```
/** @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:

```
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}`
}

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:

```
import Image from 'next/image'

export default function Page() {
  return <Image alt="turtles" src="/turtles.jpg" width={300} height={300} />
}

import Image from 'next/image'

export default function Page() {
  return <Image alt="turtles" src="/turtles.jpg" width={300} height={300} />
}
```

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:

```
export async function GET() {
  return Response.json({ name: 'Lee' })
}

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 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:

```
'use client';

import { useEffect } from 'react';

export default function ClientComponent() {
  useEffect(() => {
    // You now have access to `window`
    console.log(window.innerHeight);
  }, [])

  return ...;
}
```

Unsupported Features

Features that require a Node.js server, or dynamic logic that cannot be computed during the build process, are **not** supported:

- Dynamic Routes with `dynamicParams: true`
- Dynamic Routes without `generateStaticParams()`
- Route Handlers that rely on Request
- Cookies
- Rewrites
- Redirects
- Headers
- Middleware
- Incremental Static Regeneration
- Image Optimization with the default `loader`
- Draft Mode

Attempting to use any of these features with `next dev` will result in an error, similar to setting the `dynamic` option to `error` in the root layout.

```
export const dynamic = 'error'
```

- Internationalized Routing
- API Routes
- Rewrites
- Redirects
- Headers
- Middleware
- Incremental Static Regeneration
- Image Optimization with the default `loader`
- Draft Mode
- `getStaticPaths` with `fallback: true`
- `getStaticPaths` with `fallback: 'blocking'`
- `getServerSideProps`

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:

```
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"`

title: Deploying description: Learn how to deploy your Next.js app to production, either managed or self-hosted.

{/* 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. */}

Congratulations, it's time to ship to production.

You can deploy managed Next.js 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 to help achieve the best performance and support all modern 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.

Managed Next.js with Vercel

Vercel, 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 or deploy a template for free to try it out.

Self-Hosting

You can self-host Next.js in three different ways:

- A Node.js server
- A Docker container
- A static 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:

```
{
  "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 containers. You can use this approach when deploying to container orchestrators such as Kubernetes or when running inside a container in any cloud provider.

1. Install Docker on your machine
2. Clone our example (or the multi-environment example)
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, 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 does not support Next.js features that require a server. Learn more.

> **Good to know:**
>
> - Server Components are supported with static exports.

Features

Image Optimization

Image Optimization 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.

Image Optimization can be used with a static export 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 in your production environment by running `npm install sharp` in your project directory. On Linux platforms, `sharp` may require additional configuration to prevent excessive memory usage.
- Learn more about the caching behavior of optimized images and how to configure the TTL.
- You can also disable Image Optimization and still retain other benefits of using `next/image` if you prefer. For example, if you are optimizing images yourself separately.

Middleware

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.

Middleware uses a runtime 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 as a Server Component. For example, checking headers and redirecting. You can also use headers, cookies, or query parameters to redirect or rewrite through `next.config.js`. If that does not work, you can also use a 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. 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.

```
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.
- We do not recommend using the runtimeConfig option, as this does not work with the standalone output mode. Instead, we recommend incrementally adopting 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. You can configure the TTL 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 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.

Static Assets

If you want to host static assets on a different domain or CDN, you can use the `assetPrefix` configuration 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`.

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:

```
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:

```
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 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`:

```
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 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 for Next.js applications to ensure assets and functions from the previous build are still available while the new build is being deployed.

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`.

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "NEXT_MANUAL_SIG_HANDLE=true next start"
  }
}
```

```
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)
  })
}
```

---

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:

```
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`

```
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.

For example:

```
import { ImageResponse } from 'next/server'
```

Transforms into:

```
import { ImageResponse } from 'next/og'
```

Use `viewport` export

`metadata-to-viewport-export`

```
npx @next/codemod@latest metadata-to-viewport-export .
```

This codemod migrates certain viewport metadata to `viewport` export.

For example:

```
export const metadata = {
  title: 'My App',
  themeColor: 'dark',
  viewport: {
    width: 1,
  },
}
```

Transforms into:

```
export const metadata = {
  title: 'My App',
}

export const viewport = {
  width: 1,
  themeColor: 'dark',
}
```

13.2

Use Built-in Font

`built-in-next-font`

```
npx @next/codemod@latest built-in-next-font .
```

This codemod uninstalls the `@next/font` package and transforms `@next/font` imports into the built-in `next/font`.

For example:

```
import { Inter } from '@next/font/google'
```

Transforms into:

```
import { Inter } from 'next/font/google'
```

13.0

Rename Next Image Imports

`next-image-to-legacy-image`

```
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:

```
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:

```
// '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`

```
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`

```
npx @next/codemod@latest new-link .
```

Remove `<a>` tags inside Link Components, or add a `legacyBehavior` prop to Links that cannot be auto-fixed.

Remove `<a>` tags inside Link Components, or add a `legacyBehavior` prop to Links that cannot be auto-fixed.

For example:

```
<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.

```
const Component = () => <a>About</a>

<Link href="/about">
  <Component />
</Link>
// becomes
<Link href="/about" legacyBehavior>
  <Component />
</Link>
```

11

Migrate from CRA

`cra-to-next`

```
npx @next/codemod cra-to-next
```

Migrates a Create React App project to Next.js; creating a Pages Router and necessary config to match behavior. Client-side only rendering is leveraged initially to prevent breaking compatibility due to `window` usage during SSR and can be enabled seamlessly to allow the gradual adoption of Next.js specific features.

Please share any feedback related to this transform [in this discussion](#).

10

Add React imports

`add-missing-react-import`

```
npx @next/codemod add-missing-react-import
```

Transforms files that do not import `React` to include the import in order for the new [React JSX transform](#) to work.

For example:

```
export default class Home extends React.Component {
  render() {
    return <div>Hello World</div>
  }
}
```

Transforms into:

```
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`

```
npx @next/codemod name-default-component
```

Versions 9 and above.

Transforms anonymous components into named components to make sure they work with [Fast Refresh](#).

For example:

```
export default function () {
  return <div>Hello World</div>
}
```

Transforms into:

```
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`

```
npx @next/codemod withamp-to-config
```

Transforms the `withAmp` HOC into Next.js 9 page configuration.

For example:

```
// Before
import { withAmp } from 'next/amp'

function Home() {
  return <h1>My AMP Page</h1>
}

export default withAmp(Home)

// After
export default function Home() {
  return <h1>My AMP Page</h1>
}

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

6

Use `withRouter`

`url-to-withrouter`

```
npx @next/codemod url-to-withrouter
```

Transforms the deprecated automatically injected `url` property on top level pages to using `withRouter` and the `router` property it injects. Read more here: [https://nextjs.org/docs/messages/url-deprecated](#)

For example:

```
import React from 'react'
export default class extends React.Component {
  render() {
    const { pathname } = this.props.url
    return <div>Current pathname: {pathname}</div>
  }
}

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.

---

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
- Upgrade features that work in both the `pages` and the `app` directories
- Incrementally migrate your existing application from `pages` to `app`

Upgrading

Node.js Version

The minimum Node.js version is now **v18.17**. See the Node.js documentation for more information.

Next.js Version

To update to Next.js version 13, run the following command using your preferred package manager:

```
npm install next@latest react@latest react-dom@latest
```

ESLint Version

If you're using ESLint, you need to upgrade your ESLint version:

```
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: A guide to help you upgrade to new features such as the improved Image and Link Components.
- Migrate from the `pages` to `app` directory: A step-by-step guide to help you incrementally migrate from the `pages` to the `app` directory.

Upgrading New Features

Next.js 13 introduced the new App Router with new features and conventions. The new Router is available in the `app` directory and co-exists with the `pages` directory.

Upgrading to Next.js 13 does **not** require using the new App Router. You can continue using `pages` with new features that work in both directories, such as the updated Image component, Link component, Script component, and Font optimization.

`<Image/>` Component

Next.js 12 introduced new improvements to the Image Component with a temporary import: `next/future/image`. These improvements included less client-side JavaScript, easier ways to extend and style images, better accessibility, and native browser lazy loading.

In version 13, this new behavior is now the default for `next/image`.

There are two codemods to help you migrate to the new Image Component:

- **`next-image-to-legacy-image` codemod**: Safely and automatically renames `next/image` imports to `next/legacy/image`. Existing components will maintain the same behavior.
- **`next-image-experimental` codemod**: Dangerously adds inline styles and removes unused props. This will change the behavior of existing components to match the new defaults. To use this codemod, you need to run the `next-image-to-legacy-image` codemod first.

`<Link>` Component

The `<Link>` Component no longer requires manually adding an `<a>` tag as a child. This behavior was added as an experimental option in version 12.2 and is now the default. In Next.js 13, `<Link>` always renders `<a>` and allows you to forward props to the underlying tag.

For example:

```
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.

`<Script>` Component

The behavior of `next/script` has been updated to support both `pages` and `app`, but some changes need to be made to ensure a smooth migration:

- Move any `beforeInteractive` scripts you previously included in `_document.js` to the root layout file (`app/layout.tsx`).
- The experimental `worker` strategy does not yet work in `app` and scripts denoted with this strategy will either have to be removed or modified to use a different strategy (e.g. `lazyOnload`).
- `onLoad`, `onReady`, and `onError` handlers will not work in Server Components so make sure to move them to a Client Component or remove them altogether.

Font Optimization

Previously, Next.js helped you optimize fonts by inlining font CSS. Version 13 introduces the new `next/font` module which gives you the ability to customize your font loading experience while still ensuring great performance and privacy. `next/font` is supported in both the `pages` and `app` directories.

While inlining CSS still works in `pages`, it does not work in `app`. You should use `next/font` instead.

See the Font Optimization page to learn how to use `next/font`.

Migrating from `pages` to `app`

> **Watch:** Learn how to incrementally adopt the App Router → YouTube (16 minutes).

Moving to the App Router may be the first time using React features that Next.js builds on top of such as Server Components, Suspense, and more. When combined with new Next.js features such as special files and layouts, migration means new concepts, mental models, and behavioral changes to learn.

We recommend reducing the combined complexity of these updates by breaking down your migration into smaller steps. The `app` directory is intentionally designed to work simultaneously with the `pages` directory to allow for incremental page-by-page migration.

- The `app` directory supports nested routes *and* layouts. Learn more.
- Use nested folders to define routes and a special `page.js` file to make a route segment publicly accessible. Learn more.
- Special file conventions are used to create UI for each route segment. The most common special files are `page.js` and `layout.js`.
  - Use `page.js` to define UI unique to a route.
  - Use `layout.js` to define UI that is shared across multiple routes.
  - `.js`, `.jsx`, or `.tsx` file extensions can be used for special files.
- You can colocate other files inside the `app` directory such as components, styles, tests, and more. Learn more.
- Data fetching functions like `getServerSideProps` and `getStaticProps` have been replaced with a new API inside `app`. `getStaticPaths` has been replaced with `generateStaticParams`.
- `pages/_app.js` and `pages/_document.js` have been replaced with a single `app/layout.js` root layout. Learn more.
- `pages/_error.js` has been replaced with more granular `error.js` special files. Learn more.
- `pages/404.js` has been replaced with the `not-found.js` file.
- `pages/api/*` currently remain inside the `pages` directory.

Step 1: Creating the `app` directory

Update to the latest Next.js version (requires 13.4 or greater):

```
npm install next@latest
```

Then, create a new `app` directory at the root of your project (or `src/` directory).

Step 2: Creating a Root Layout

Create a new `app/layout.tsx` file inside the `app` directory. This is a root layout that will apply to all routes inside `app`.

```
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>
  )
}
```

```
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:

```
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'Home',
  description: 'Welcome to Next.js',
}

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.

Migrating the `getLayout()` pattern to Layouts (Optional)

Next.js recommended adding a property to Page components to achieve per-page layouts in the `pages` directory. This pattern can be replaced with native support for nested layouts in the `app` directory.

▶ See before and after example

Step 3: Migrating `next/head`

In the `pages` directory, the `next/head` React component is used to manage `<head>` HTML elements such as `title` and `meta` . In the `app` directory, `next/head` is replaced with the new built-in SEO support.

Before:

```
import Head from 'next/head'

export default function Page() {
  return (
    <>
      <Head>
        <title>My page title</title>
      </Head>
    </>
  )
}
```

```
import Head from 'next/head'

export default function Page() {
  return (
    <>
      <Head>
        <title>My page title</title>
      </Head>
    </>
  )
}
```

After:

```
import { Metadata } from 'next'

export const metadata: Metadata = {
  title: 'My Page Title',
}

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

```
export const metadata = {
  title: 'My Page Title',
}

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

See all metadata options.

Step 4: Migrating Pages

- Pages in the `app` directory are Server Components by default. This is different from the `pages` directory where pages are Client Components.
- Data fetching has changed in `app`. `getServerSideProps`, `getStaticProps` and `getInitialProps` have been replaced with a simpler API.
- The `app` directory uses nested folders to define routes and a special `page.js` file to make a route segment publicly accessible.
`pages` Directory	`app` Directory	Route
`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`.

```
'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>
  )
}
```


```
'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. See the data fetching upgrade guide for more details.

```
// 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} />
}
```


```
// 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.
- Start your development server and visit `http://localhost:3000`. You should see your existing index route, now served through the app directory.

Step 5: Migrating Routing Hooks

A new router has been added to support the new behavior in the `app` directory.

In `app`, you should use the three new hooks imported from `next/navigation`: `useRouter()`, `usePathname()`, and `useSearchParams()`.

- The new `useRouter` hook is imported from `next/navigation` and has different behavior to the `useRouter` hook in `pages` which is imported from `next/router`.
  - The `useRouter` hook imported from `next/router` is not supported in the `app` directory but can continue to be used in the `pages` directory.
- The new `useRouter` does not return the `pathname` string. Use the separate `usePathname` hook instead.
- The new `useRouter` does not return the `query` object. Use the separate `useSearchParams` hook instead.
- You can use `useSearchParams` and `usePathname` together to listen to page changes. See the Router Events section for more details.
- These new hooks are only supported in Client Components. They cannot be used in Server Components.

```
'use client'

import { useRouter, usePathname, useSearchParams } from 'next/navigation'

export default function ExampleClientComponent() {
  const router = useRouter()
```

```
  const pathname = usePathname()
  const searchParams = useSearchParams()

  // ...
}

'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.
- The `locale`, `locales`, `defaultLocales`, `domainLocales` values have been removed because built-in i18n Next.js features are no longer necessary in the `app` directory. Learn more about i18n.
- `basePath` has been removed. The alternative will not be part of `useRouter`. It has not yet been implemented.
- `asPath` has been removed because the concept of `as` has been removed from the new router.
- `isReady` has been removed because it is no longer necessary. During static rendering, any component that uses the `useSearchParams()` hook will skip the prerendering step and instead be rendered on the client at runtime.

View the `useRouter()` API reference.

Step 6: Migrating Data Fetching Methods

The `pages` directory uses `getServerSideProps` and `getStaticProps` to fetch data for pages. Inside the `app` directory, these previous data fetching functions are replaced with a simpler API built on top of `fetch()` and `async` React Server Components.

```
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>
}
```

```
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).

```
// `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. This allows us to send less JavaScript to the client, while maintaining the rendered HTML from the server.

By setting the `cache` option to `no-store`, we can indicate that the fetched data should never be cached. This is similar to `getServerSideProps` in the `pages` directory.

```
// `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>
  )
}

// `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.

```
// `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()`: Based on the Web Headers API, and can be used inside Server Components to retrieve request headers.
- `cookies()`: Based on the Web Cookies API, and can be used inside Server Components to retrieve cookies.

```
// `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 '...'
}

// `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.

```
// `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()` will default to `cache: 'force-cache'`, which will cache the request data until manually invalidated. This is similar to `getStaticProps` in the `pages` directory.

```
// `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.

```
// `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`.

`generateStaticParams` behaves similarly to `getStaticPaths`, but has a simplified API for returning route parameters and can be used inside layouts. The return shape of `generateStaticParams` is an array of segments instead of an array of nested `param` objects or a string of resolved paths.

```
// `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.

```
// `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 controls how params outside of `generateStaticParams` are handled:

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

This replaces the `fallback: true | false | 'blocking'` option of `getStaticPaths` in the `pages` directory. The `fallback: 'blocking'` option is not included in `dynamicParams` because the difference between `'blocking'` and `true` is negligible with streaming.

```
// `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` 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.

```
// `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()` can use `revalidate`, which will cache the request for the specified amount of seconds.

```
// `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 in the `app` directory.

Route Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.

```
export async function GET(request: Request) {}

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 instead to securely fetch data. Learn more about data fetching.

Step 7: Styling

In the `pages` directory, global stylesheets are restricted to only `pages/_app.js`. With the `app` directory, this restriction has been lifted. Global styles can be added to any layout, page, or component.

- CSS Modules
- Tailwind CSS
- Global Styles
- CSS-in-JS
- External Stylesheets
- Sass

Tailwind CSS

If you're using Tailwind CSS, you'll need to add the `app` directory to your `tailwind.config.js` file:

```
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:

```
import '../styles/globals.css'

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

Learn more about styling with Tailwind CSS

Codemods

Next.js provides Codemod transformations to help upgrade your codebase when a feature is deprecated. See Codemods for more information.

---

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:

```
npm i next@latest react@latest react-dom@latest eslint-config-next@latest

yarn add next@latest react@latest react-dom@latest eslint-config-next@latest

pnpm up next react react-dom eslint-config-next --latest

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 for more information.
- The `next/server` import for `ImageResponse` was renamed to `next/og`. A codemod is available 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 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:

Slow initial page loading time

If you have built your application with the default Vite plugin for 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.

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.

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. With Next.js, this issue is resolved by fetching data in Server Components.

Fast and intentional loading states

Thanks to built-in support for 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.

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.

Middleware

Next.js 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.

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:

```
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.

```
/** @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 to `tsconfig.node.json`
2. Add `./dist/types/**/*.ts` and `./next-env.d.ts` to the `include` array
3. Add `./node_modules` to the `exclude` array
4. Add `{ "name": "next" }` to the `plugins` array in `compilerOptions`: `"plugins": [{ "name": "next" }]`
5. Set `esModuleInterop` to `true`: `"esModuleInterop": true`
6. Set `jsx` to `preserve`: `"jsx": "preserve"`
7. Set `allowJs` to `true`: `"allowJs": true`
8. Set `forceConsistentCasingInFileNames` to `true`: `"forceConsistentCasingInFileNames": true`
9. Set `incremental` to `true`: `"incremental": true`

Here's an example of a working `tsconfig.json` with those changes:

```
{
  "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.

Step 4: Create the Root Layout

A Next.js App Router application must include a root layout file, which is a React Server Component 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, 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:

```
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return null
}
```

```
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>`:

```
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>
  )
}
```

```
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 and meta viewport tags, so you can safely remove those from your `<head>`:

```
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>
  )
}
```

```
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 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 into the `app` directory you can safely delete their `<link>` tags:

```
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>
  )
}
```

```
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. Move your final metadata info into an exported `metadata` object:

```
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>
  )
}
```

```
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). 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. Next.js uses a file-system based router where directories are used to define 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:**

```
'use client'

import dynamic from 'next/dynamic'
import '../../index.css'

const App = dynamic(() => import('../../App'), { ssr: false })

export default function Page() {
  return <App />
}
```

```
'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 by the `'use client'` directive. Without that directive, the component would have been a Server Component.

In Next.js, Client Components are prerendered to HTML 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`:

```
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:

```
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, 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. 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:**

```
// 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:**

```
// 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 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:**

```
# ...
NEXT_PUBLIC_BASE_PATH="/some-base-path"
```

2. **Set `basePath` to `process.env.NEXT_PUBLIC_BASE_PATH` in your `next.config.mjs` file:**

```
/** @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`:

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start"
  }
}
```

```
# ...
.next
```

next-env.d.ts

Now run `npm run dev`, and open 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 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 to get:
  - Automatic code splitting
  - Streaming Server-Rendering
  - React Server Components
- Optimize images with the `<Image>` component
- Optimize fonts with `next/font`
- Optimize third-party scripts with the `<Script>` component
- Update your ESLint configuration to support Next.js rules

---

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.

If you're new to Next.js, we recommend starting with the Routing, Rendering, Data Fetching and Styling sections, as they introduce the fundamental Next.js and web concepts to help you get started. Then, you can dive deeper into the other sections such as Optimizing and Configuring. Finally, once you're ready, checkout the Deploying and Upgrading sections.

If you're new to Next.js, we recommend starting with the Routing, Rendering, Data Fetching and Styling sections, as they introduce the fundamental Next.js and web concepts to help you get started. Then, you can dive deeper into the other sections such as Optimizing and Configuring. Finally, once you're ready, checkout the Deploying and Upgrading sections.

---

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` and `next/font/local`. For features and usage, please see the Optimizing Fonts page.

Font Function Arguments

For usage, review Google Fonts and Local Fonts.

Key	`font/google`	`font/local`	Type	Required
`src`			String or Array of Objects	Yes
`weight`			String or Array	Required/Optional
`style`			String or Array	-
`subsets`			Array of Strings	-
`axes`			Array of Strings	-
`display`			String	-
`preload`			Boolean	-
`fallback`			Array of Strings	-

Key	font/google	font/local	Type	Required
adjustFontFallback			Boolean or String	-
variable			String	-
declarations			Array of Objects	-

src

The path of the font file as a string or an array of objects (with type `Array<{path: string, weight?: string, style?: string}>`) relative to the directory where the font loader function is called.

Used in `next/font/local`

- Required

Examples:

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

weight

The font `weight` with the following possibilities:

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

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

- Required if the font being used is **not** variable

Examples:

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

style

The font `style` with the following possibilities:

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

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

- Optional

Examples:

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

subsets

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

Used in `next/font/google`

- Optional

Examples:

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

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

axes

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

Used in `next/font/google`

- Optional

Examples:

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

display

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

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

- Optional

Examples:

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

`preload`

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

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

- Optional

Examples:

- `preload: false`

`fallback`

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

- Optional

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

Examples:

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

`adjustFontFallback`

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

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

- Optional

Examples:

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

`variable`

A string value to define the CSS variable name to be used if the style is applied with the CSS variable method.

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

- Optional

Examples:

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

`declarations`

An array of font face descriptor key-value pairs that define the generated `@font-face` further.

Used in `next/font/local`

- Optional

Examples:

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

Applying Styles

You can apply the font styles in three ways:

- `className`
- `style`
- CSS Variables

`className`

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

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

`style`

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

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

CSS Variables

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

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

```
import { Inter } from 'next/font/google'
import styles from '../styles/component.module.css'

const inter = Inter({
  variable: '--font-inter',
})
```

```
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.

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

```
<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:

```
.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:

```
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 }
```

```
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:

```
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>
  )
}
```

```
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:

```
{
  "compilerOptions": {
    "paths": {
      "@/fonts": ["./styles/fonts"]
    }
  }
}
```

You can now import any font definition as follows:

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


```
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: 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. */}

▶ Examples

> **Good to know**: If you are using a version of Next.js prior to 13, you'll want to use the next/legacy/image documentation since the component was renamed.

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

```
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:

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	-

Required Props

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

```
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 image file
- A path string. This can be either an absolute external URL, or an internal path depending on the loader prop.

When using an external URL, you must add it to remotePatterns in `next.config.js`.

`width`

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

Required, except for statically imported images or images with the `fill` property.

`height`

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

Required, except for statically imported images or images with the `fill` property.

`alt`

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

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

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

Learn more

Optional Props

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

`loader`

A custom function used to resolve image URLs.

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

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

Here is an example of using a custom loader:

```
'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}
    />
  )
}
```

Good to know: Using props like `loader`, which accept a function, requires using Client Components to serialize the provided function.

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

`fill`

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

A boolean that causes the image to fill the parent element, which is useful when the `width` and `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`
- `object-fit`
- `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` or which are styled to have a responsive size.

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

- First, the value of `sizes` is used by the browser to determine which size of the image to download, from `next/image`'s automatically generated `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:

```
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
- mdn

`quality`

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

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

`priority`

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

When true, the image will be considered high priority and preload. Lazy loading is automatically disabled for images using `priority`.

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

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

`placeholder`

```
placeholder = 'empty' // "empty" | "blur" | "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` property will be used as the placeholder. If `src` is an object from a static import and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated, except when the image is detected to be animated.

For dynamic images, you must provide the `blurDataURL` property. Solutions such as Plaiceholder can help with `base64` generation.

When `data:image/...`, the Data URL 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
- Demo the shimmer effect with data URL `placeholder` prop
- Demo the color effect with `blurDataURL` prop

Advanced Props

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

`style`

Allows passing CSS styles to the underlying image element.

```
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`

```
'use client'

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

> **Warning**: Deprecated since Next.js 14 in favor of `onLoad`.

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

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

> **Good to know**: Using props like `onLoadingComplete`, which accept a function, requires using Client Components to serialize the provided function.

`onLoad`

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

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

The callback function will be called with one argument, the Event which has a `target` that references the underlying `<img>` element.

> **Good to know**: Using props like `onLoad`, which accept a function, requires using Client Components to serialize the provided function.

`onError`

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

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

> **Good to know**: Using props like `onError`, which accept a function, requires using Client Components to serialize the provided function.

`loading`

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

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

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

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

When `eager`, load the image immediately.

Learn more about the `loading` attribute.

`blurDataURL`

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

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

Try it out:

- Demo the default `blurDataURL` prop
- Demo the color effect with `blurDataURL` prop

You can also generate a solid color Data URL to match the image.

`unoptimized`

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

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

```
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:

```
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 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:

```
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:

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
        port: '',
      },
    ],
  },
}
```

> **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, port, 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.

> **Good to know**: When omitting `protocol`, `port` or `pathname`, then the wildcard `**` is implied. This is not recommended because it may allow malicious actors to optimize urls you did not intend.

`domains`

> **Warning**: Deprecated since Next.js 14 in favor of strict `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`, 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:

```
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:

```
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:

```
'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 to configure each instance of `next/image`.

Examples:

- Custom Image Loader Configuration

> **Good to know**: Customizing the image loader file, which accepts a function, requires using Client Components to serialize the provided function.

Advanced

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

`deviceSizes`

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

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

```
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 to form the full array of sizes used to generate image srcsets.

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

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

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

`formats`

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

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

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

```
module.exports = {
  images: {
    formats: ['image/webp'],
  },
}
```

You can enable AVIF support with the following configuration.

```
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. For all other loaders, please refer to your cloud provider's documentation.

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

The cache status of an image can be determined by reading the value of the `x-nextjs-cache` response header. The possible values are the following:

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

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

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

`minimumCacheTTL`

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

```
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` to set the `Cache-Control` header on the upstream image (e.g. `/some-asset.jpg`, not `/_next/image` itself).

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

`disableStaticImages`

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

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

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

```
module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

`dangerouslyAllowSVG`

The default loader does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without a proper 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`:

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

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

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` 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:

```
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

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:

```
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

Responsive image with `fill`

If you don't know the aspect ratio, you will need to set the `fill` prop and set `position: relative` on the parent. Optionally, you can set `object-fit` style depending on the desired stretch vs crop behavior:

```
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

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.

```
.imgDark {
  display: none;
}

@media (prefers-color-scheme: dark) {
  .imgLight {
    display: none;
  }
  .imgDark {
    display: unset;
  }
}
```

```
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} />
    </>
  )
}
```

```
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"`.

Try it out:

- Demo light/dark mode theme detection

Known Browser Bugs

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

- Safari 15 - 16.3 display a gray border while loading. Safari 16.4 fixed this issue. Possible solutions:
  - Use CSS `@supports (font: -apple-system-body) and (-webkit-appearance: none) { img[loading="lazy"] { clip-path: inset(0.6px) } }`
  - Use `priority` if the image is above the fold
- Firefox 67+ displays a white background while loading. Possible solutions:

- Enable AVIF `formats`
- Use `placeholder`

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 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. AVIF support added. Wrapper `<div>` changed to `<span>`.
`v11.1.0`	`onLoadingComplete` and `lazyBoundary` props added.
`v11.0.0`	`src` prop support for static import. `placeholder` prop added. `blurDataURL` prop added.
`v10.0.5`	`loader` prop added.
`v10.0.1`	`layout` prop added.
`v10.0.0`	`next/image` introduced.

---

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: 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. */}

▶ Examples

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

```
import Link from 'next/link'

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

```
import Link from 'next/link'

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

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:

```
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
```

Props

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

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

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

`href` (required)

The path or URL to navigate to.

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

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

```
// 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 stack.

```
import Link from 'next/link'

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

```
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.

```
import Link from 'next/link'

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

```
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.

```
import Link from 'next/link'

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

```
import Link from 'next/link'

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

Other Props

`legacyBehavior`

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

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

`passHref`

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

`scroll`

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

`shallow`

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

`locale`

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

Examples

Linking to Dynamic Routes

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

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

```
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
```

For example, you can generate a list of links to the dynamic route `app/blog/[slug]/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>
  )
}
```

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

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

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

```
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'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`:

```
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:

```
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: `/blog/my-post`

You can use every property as defined in the Node.js URL module documentation.

Replace the URL instead of push

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

```
<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`:

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

Middleware

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

For example, if you 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:

```
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:

```
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>
  )
}
```

> **Good to know**: If you're using Dynamic Routes, you'll need to adapt your `as` and `href` props. For example, if you have a Dynamic Route like `/dashboard/[user]` that you want to present differently via middleware, you would write: `<Link href={{ pathname: '/dashboard/authed/[user]', query: { user: username } }} as="/dashboard/[user]">Profile</Link>`.

Version History

Version	Changes
`v13.0.0`	No longer requires a child `<a>` tag. A codemod 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: