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 18.18 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 your code inside a `src/` directory? No / Yes
Would you like to use App Router? (recommended) No / Yes
Would you like to use Turbopack for `next dev`?  No / Yes
Would you like to customize the import alias (`@/*` by default)? 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.

If you're new to Next.js, see the project structure docs for an overview of all the possible files and folders in your application.

> **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.

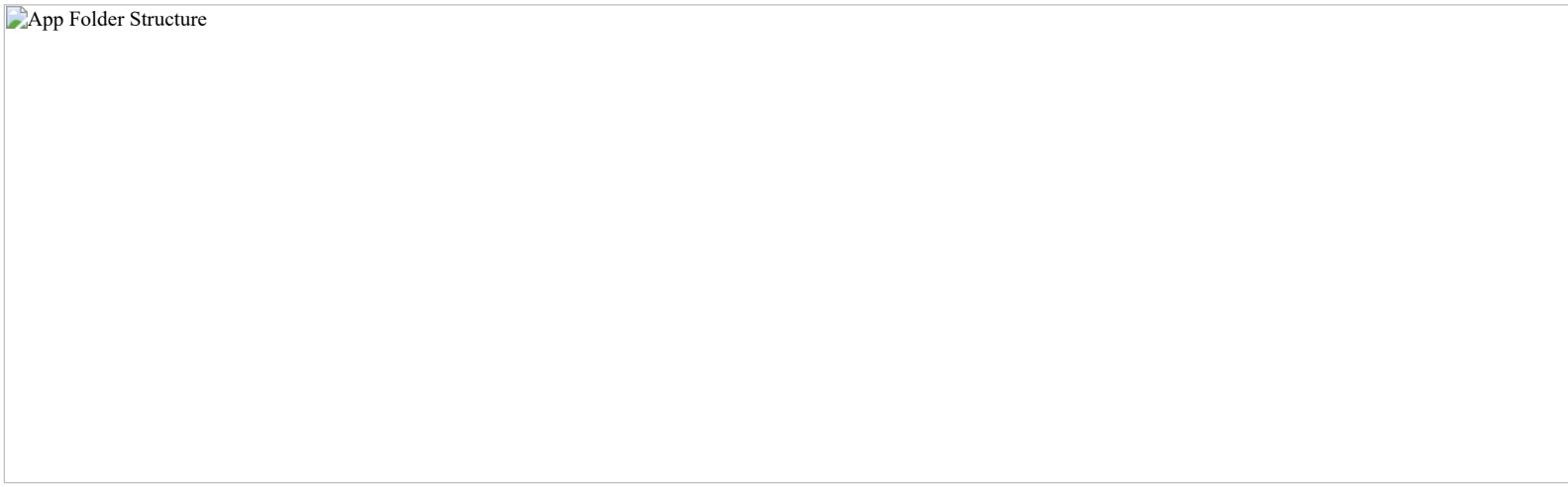
Creating directories

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>
}
```

```
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>
  )
}
```

```
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.

---

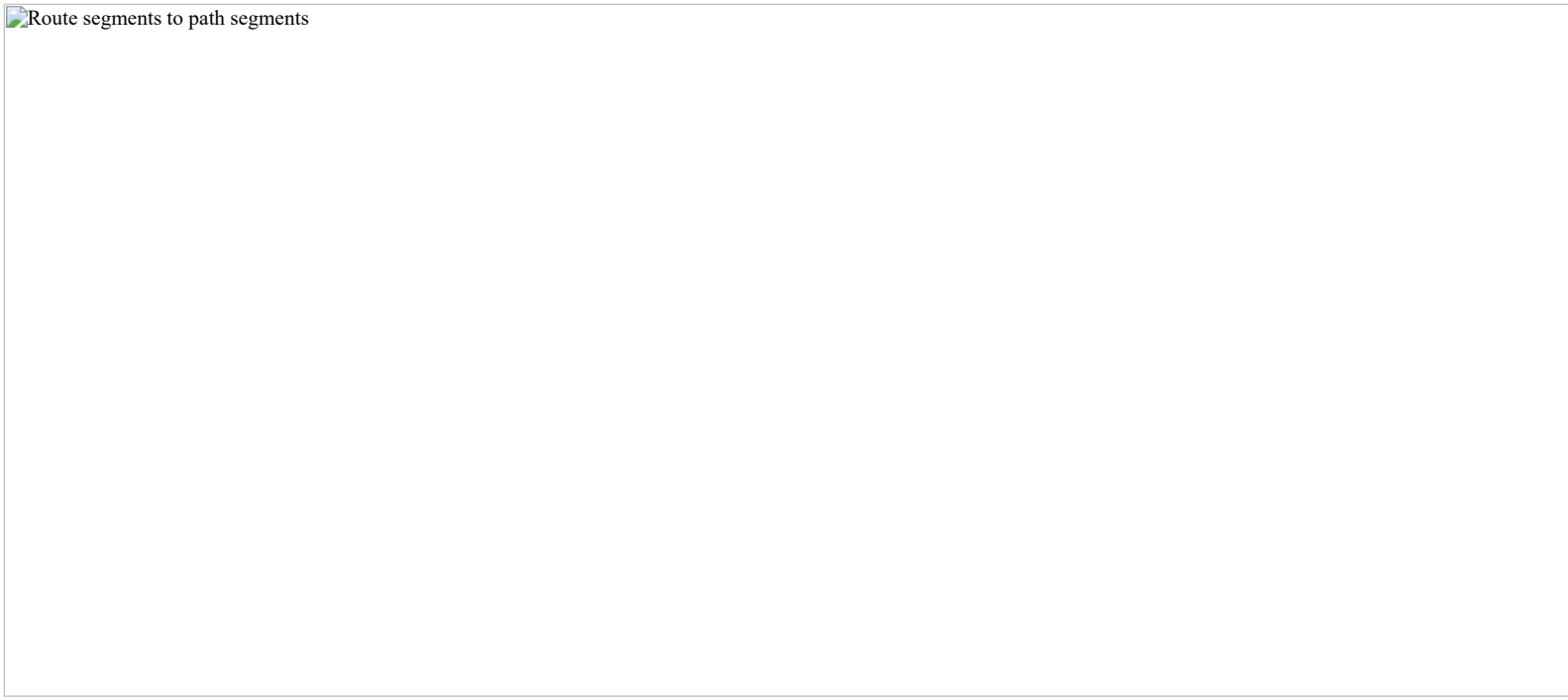
title: Next.js Project Structure nav_title: Project Structure description: A list of folders and files conventions in a Next.js project

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

Click the file and folder names to learn more about each convention.

Top-level folders

Top-level folders are used to organize your application's code and static assets.



`app`	App Router
`pages`	Pages Router
`public`	Static assets to be served
`src`	Optional application source folder

Top-level files

Top-level files are used to configure your application, manage dependencies, run middleware, integrate monitoring tools, and define environment variables.

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

The following file conventions are used to define routes and handle metadata in the `app` router.

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

The following file conventions are used to define routes in the pages router.

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

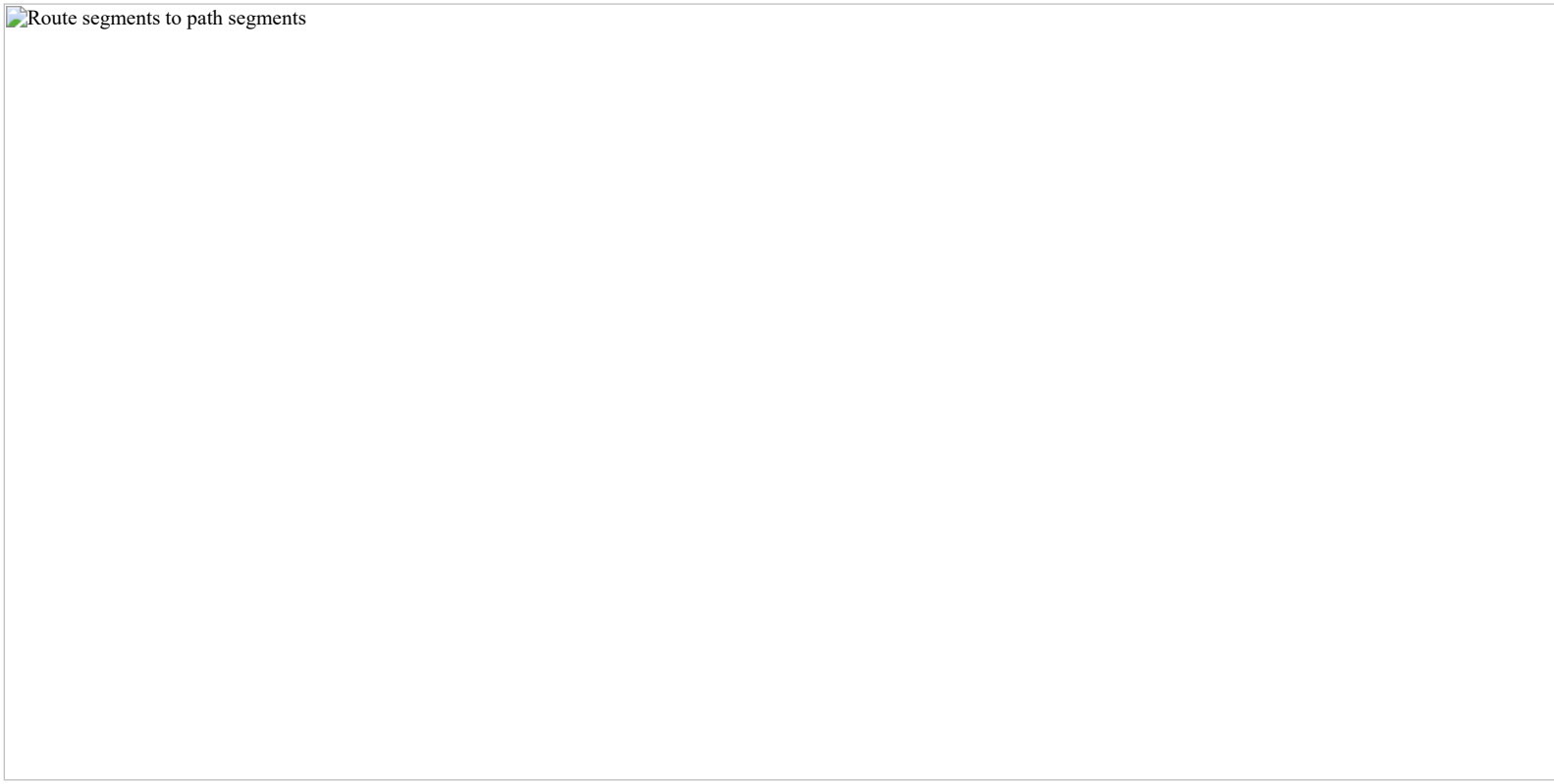
> We recommend reading the Routing Fundamentals page before continuing.

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

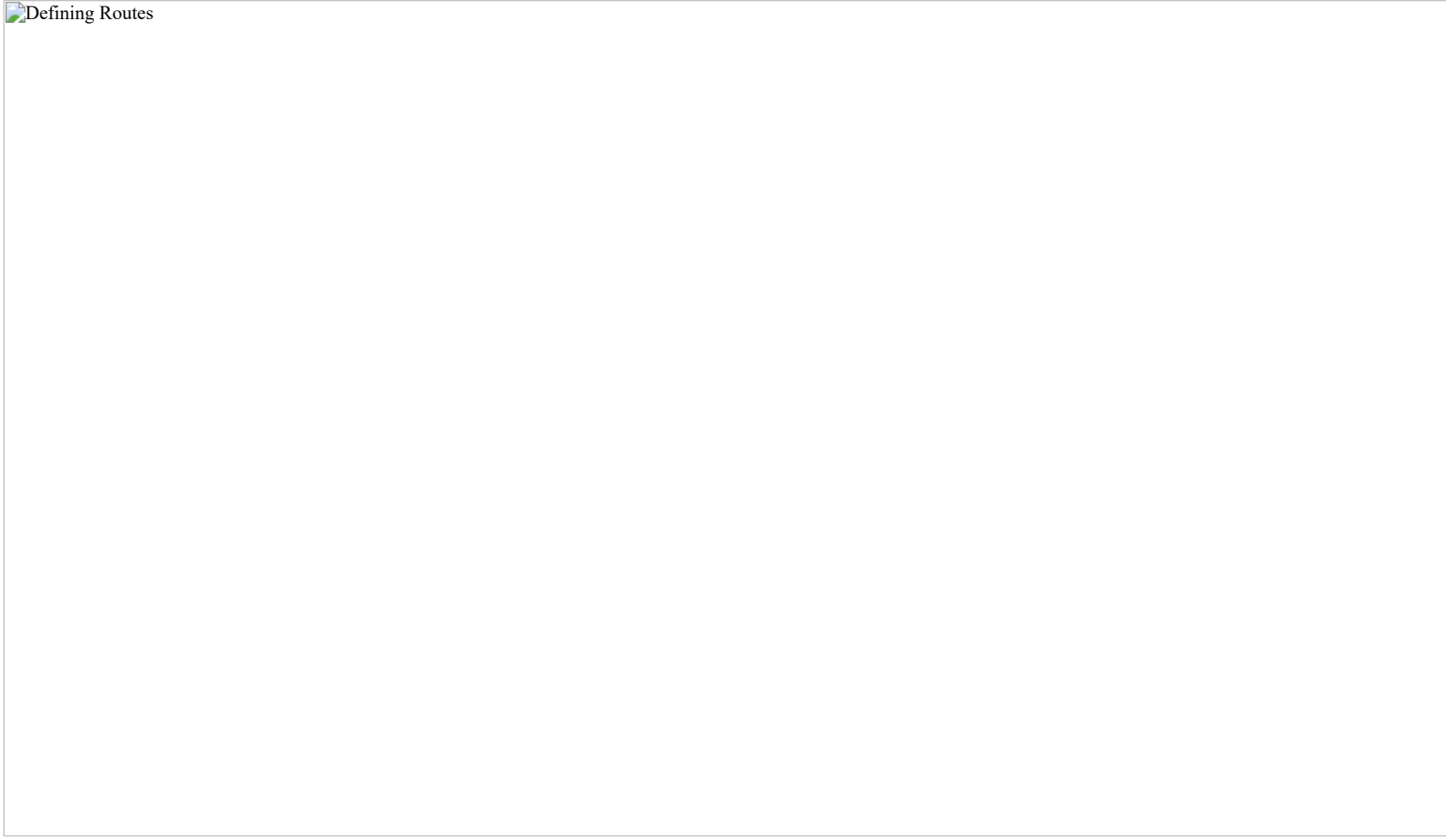
Creating Routes

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`, `.ts`, 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>
}
```

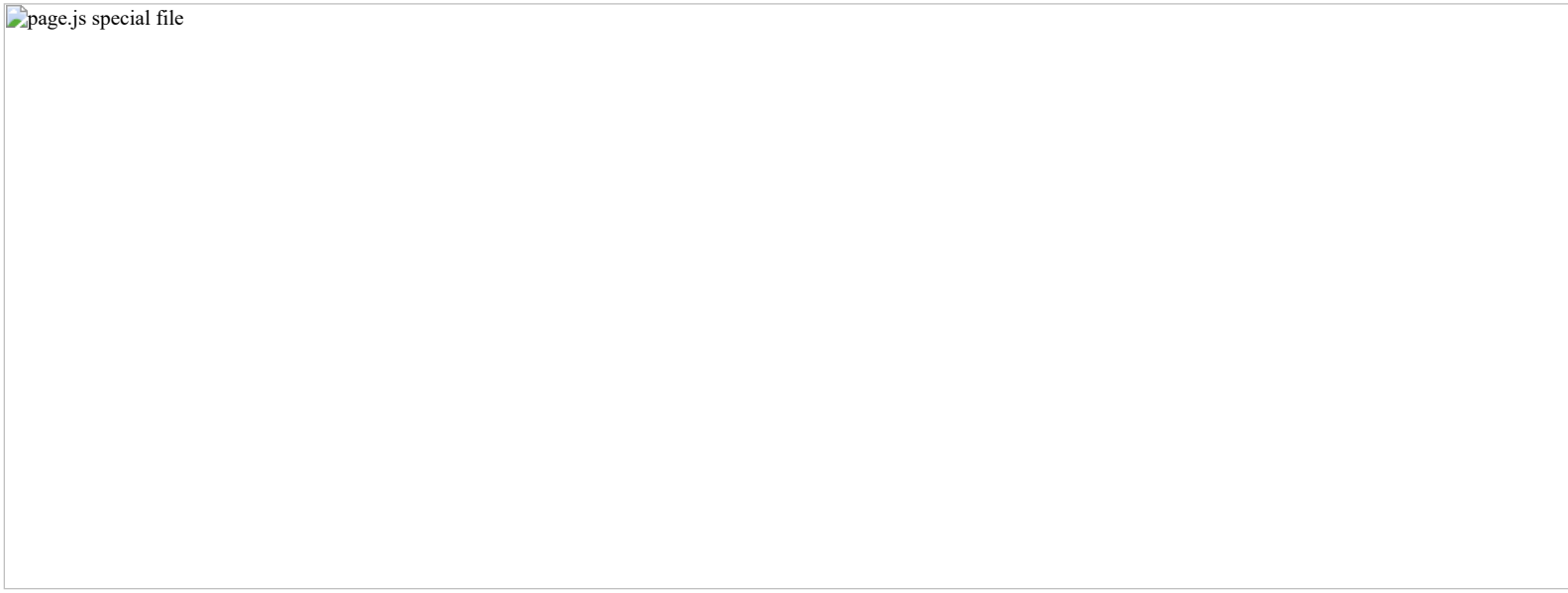

```
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```


---

title: Pages description: Create your first page in Next.js related: links: - app/building-your-application/routing/layouts-and-templates - app/building-your-application/routing/linking-and-navigating

A page is UI that is **unique** to a route. You can define a page by default exporting a component from a `page.js` file.

For example, to create your `index` page, add the `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>
}
```

Then, to create further pages, create a new folder and add the `page.js` file inside it. For example, to create a page for the `/dashboard` route, create a new folder called `dashboard`, and add the `page.js` file inside it:

```
// `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**:
>
> - The `.js`, `.jsx`, or `.tsx` file extensions can be used for Pages.
> - A page is always the leaf of the route subtree.
> - 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.

---

title: Layouts and Templates description: Create your first shared layout in Next.js.

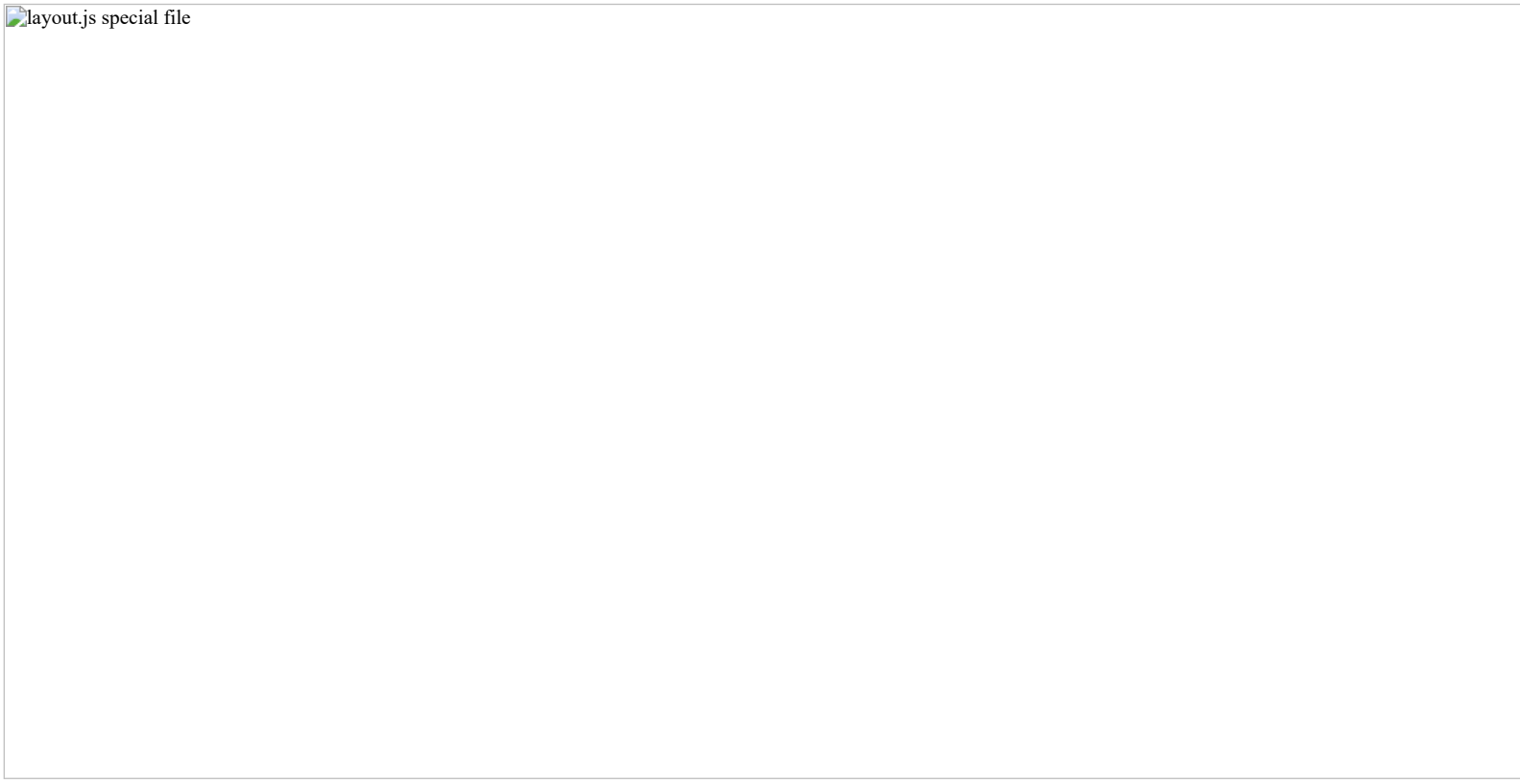
The special files layout.js and template.js allow you to create UI that is shared between routes. This page will guide you through how and when to use these special files.

Layouts

A layout is UI that is **shared** between multiple routes. 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 page during rendering.

For example, the layout will be shared with the `/dashboard` and `/dashboard/settings` pages:




```
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>
  )
}
```

Root Layout (Required)

The root layout is defined at the top level of the `app` directory and applies to all routes. This layout is **required** and must contain `html` and `body` tags, allowing you to modify the initial HTML returned from the server.

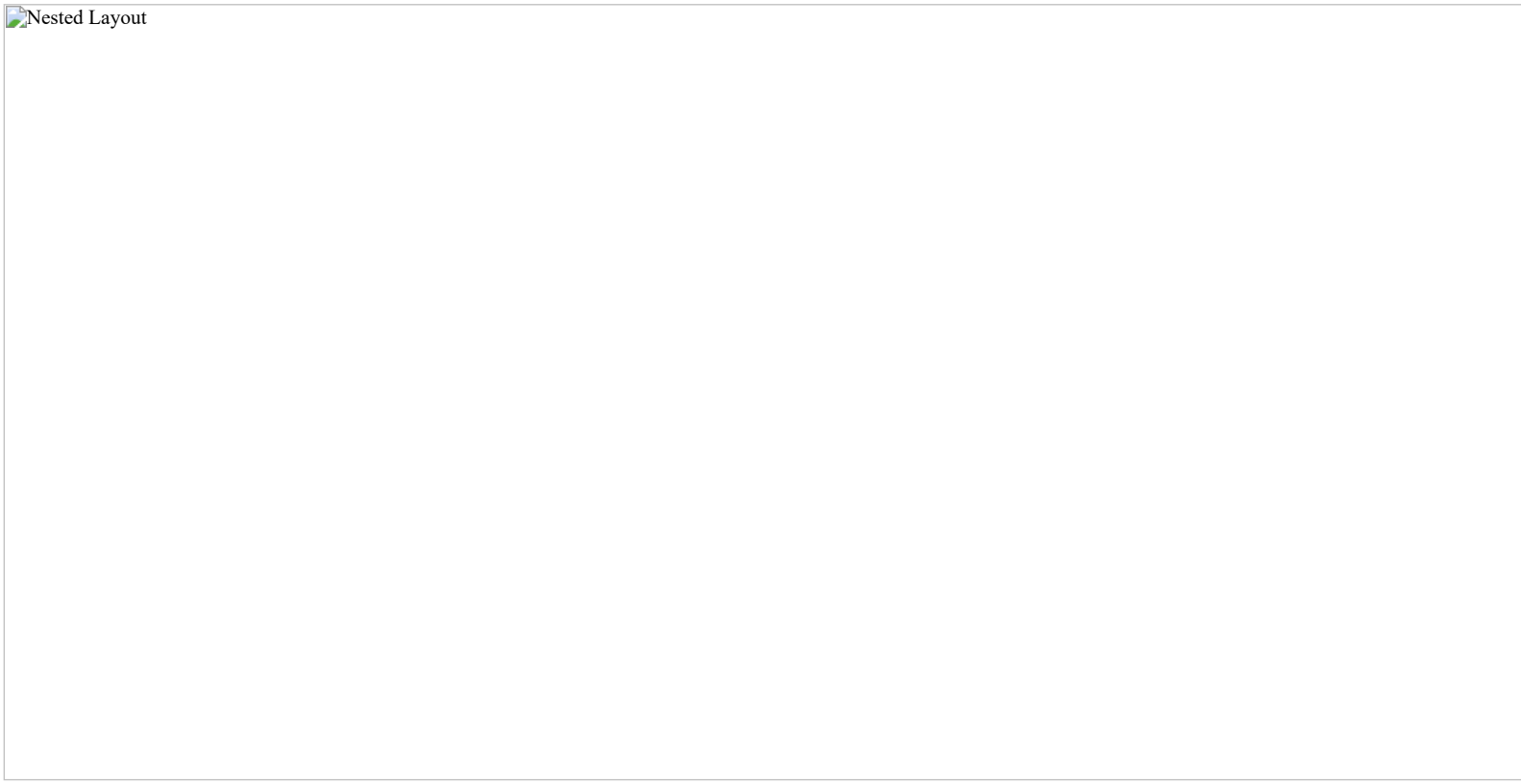
```
export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html lang="en">
      <body>
        {/* Layout UI */}
        <main>{children}</main>
      </body>
    </html>
  )
}
```

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

Nesting Layouts

By default, layouts in the folder hierarchy are **nested**, which means they wrap child layouts via their `children` prop. You can nest layouts by adding `layout.js` inside specific route segments (folders).

For example, to create a layout for the `/dashboard` route, add a new `layout.js` file inside the `dashboard` folder:




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

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

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:

Good to know:

- `.js`, `.jsx`, or `.tsx` file extensions can be used for Layouts.
- Only the root layout can contain `<html>` and `<body>` tags.
- When a `layout.js` and `page.js` file are defined in the same folder, the layout will wrap the page.
- 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 `pathname` (learn more). But imported Client Components can access the pathname using `usePathname` hook.
- 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.
- You can use Route Groups to opt specific route segments in and out of shared layouts.
- You can use Route Groups to create multiple root layouts. See an example here.
- **Migrating from the `pages` directory:** The root layout replaces the `_app.js` and `_document.js` files. View the migration guide.

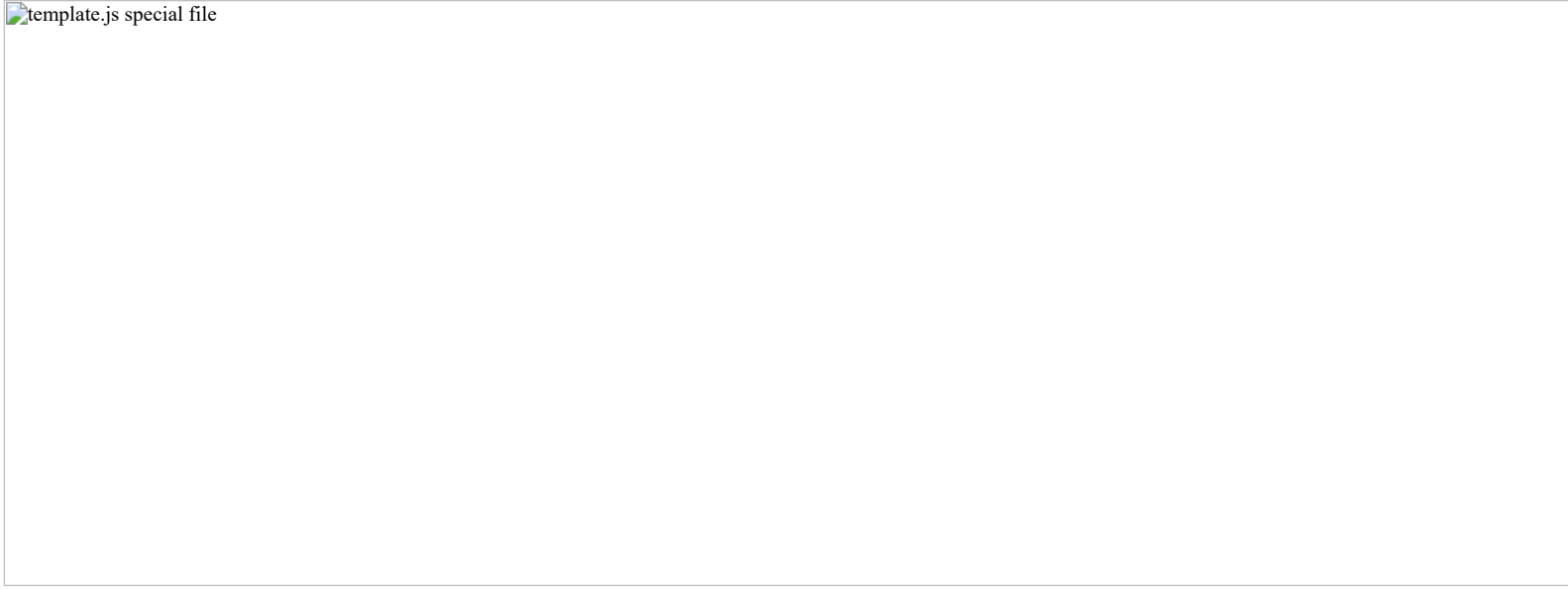
Templates

Templates are similar to layouts in that they wrap a 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 child is mounted, DOM elements are recreated, state is **not** preserved in Client Components, 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:

- To resynchronize `useEffect` on navigation.
- To reset the state of a child Client Components on 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>
```

Examples

Metadata

You can modify the `<head>` HTML elements such as `title` and `meta` using the Metadata APIs.

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

```
import type { 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, 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.

Active Nav Links

You can use the usePathname() hook to determine if a nav link is active.

Since `usePathname()` is a client hook, you need to extract the nav links into a Client Component, which can be imported into your layout or template:

```
'use client'

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

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

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

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

```
'use client'

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

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

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

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

```
import { NavLinks } from '@/app/ui/nav-links'

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

```
import { NavLinks } from '@/app/ui/nav-links'

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

---

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 four ways to navigate between routes in Next.js:

- Using the `<Link>` Component
- Using the `useRouter` hook (Client Components)
- Using the `redirect` function (Server Components)
- Using the native History API

This page will go through how to use each of these options, 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 and recommended 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.

`useRouter()` hook

The `useRouter` hook allows you to programmatically change routes from Client Components.

```
'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`.

`redirect` function

For Server Components, use the `redirect` function instead.

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

async function fetchTeam(id: string) {
  const res = await fetch('https://...')
  if (!res.ok) return undefined
  return res.json()
}

export default async function Profile({ params }: { params: { id: string } }) {
  const team = await fetchTeam(params.id)
  if (!team) {
    redirect('/login')
  }

  // ...
}
```

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

async function fetchTeam(id) {
  const res = await fetch('https://...')
  if (!res.ok) return undefined
  return res.json()
}

export default async function Profile({ params }) {
  const team = await fetchTeam(params.id)
  if (!team) {
    redirect('/login')
  }

  // ...
}
```

> **Good to know**:
>
> - `redirect` returns a 307 (Temporary Redirect) status code by default. When used in a Server Action, it returns a 303 (See Other), which is commonly used for redirecting to a success page as a result of a POST request.
> - `redirect` internally throws an error so it should be called outside of `try/catch` blocks.
> - `redirect` can be called in Client Components during the rendering process but not in event handlers. You can use the `useRouter` hook instead.
> - `redirect` also accepts absolute URLs and can be used to redirect to external links.
> - If you'd like to redirect before the render process, use `next.config.js` or Middleware.

See the `redirect` API reference for more information.

Using the native History API

Next.js allows you to use the native `window.history.pushState` and `window.history.replaceState` methods to update the browser's history stack without reloading the page.

`pushState` and `replaceState` calls integrate into the Next.js Router, allowing you to sync with `usePathname` and `useSearchParams`.

`window.history.pushState`

Use it to add a new entry to the browser's history stack. The user can navigate back to the previous state. For example, to sort a list of products:

```
'use client'

import { useSearchParams } from 'next/navigation'

export default function SortProducts() {
  const searchParams = useSearchParams()

  function updateSorting(sortOrder: string) {
    const params = new URLSearchParams(searchParams.toString())
    params.set('sort', sortOrder)
    window.history.pushState(null, '', `?${params.toString()}`)
  }

  return (
    <>
      <button onClick={() => updateSorting('asc')}>Sort Ascending</button>
      <button onClick={() => updateSorting('desc')}>Sort Descending</button>
    </>
  )
}
```

```
'use client'

import { useSearchParams } from 'next/navigation'

export default function SortProducts() {
  const searchParams = useSearchParams()

  function updateSorting(sortOrder) {
    const params = new URLSearchParams(searchParams.toString())
    params.set('sort', sortOrder)
    window.history.pushState(null, '', `?${params.toString()}`)
  }

  return (
    <>
      <button onClick={() => updateSorting('asc')}>Sort Ascending</button>
      <button onClick={() => updateSorting('desc')}>Sort Descending</button>
    </>
  )
}
```

`window.history.replaceState`

Use it to replace the current entry on the browser's history stack. The user is not able to navigate back to the previous state. For example, to switch the application's locale:

```
'use client'

import { usePathname } from 'next/navigation'

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

  function switchLocale(locale: string) {
    // e.g. '/en/about' or '/fr/contact'
    const newPath = `/${locale}${pathname}`
    window.history.replaceState(null, '', newPath)
  }

  return (
    <>
      <button onClick={() => switchLocale('en')}>English</button>
      <button onClick={() => switchLocale('fr')}>French</button>
    </>
  )
}
```

```
'use client'

import { usePathname } from 'next/navigation'

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

  function switchLocale(locale) {
    // e.g. '/en/about' or '/fr/contact'
    const newPath = `/${locale}${pathname}`
    window.history.replaceState(null, '', newPath)
  }

  return (
    <>
      <button onClick={() => switchLocale('en')}>English</button>
      <button onClick={() => switchLocale('fr')}>French</button>
    </>
  )
}
```

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 default prefetching behavior (i.e. when the `prefetch` prop is left unspecified or set to `null`) is different depending on your usage of `loading.js`. 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`. Alternatively, you can prefetch the full page data beyond the loading boundaries by setting the `prefetch` prop to `true`.

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.

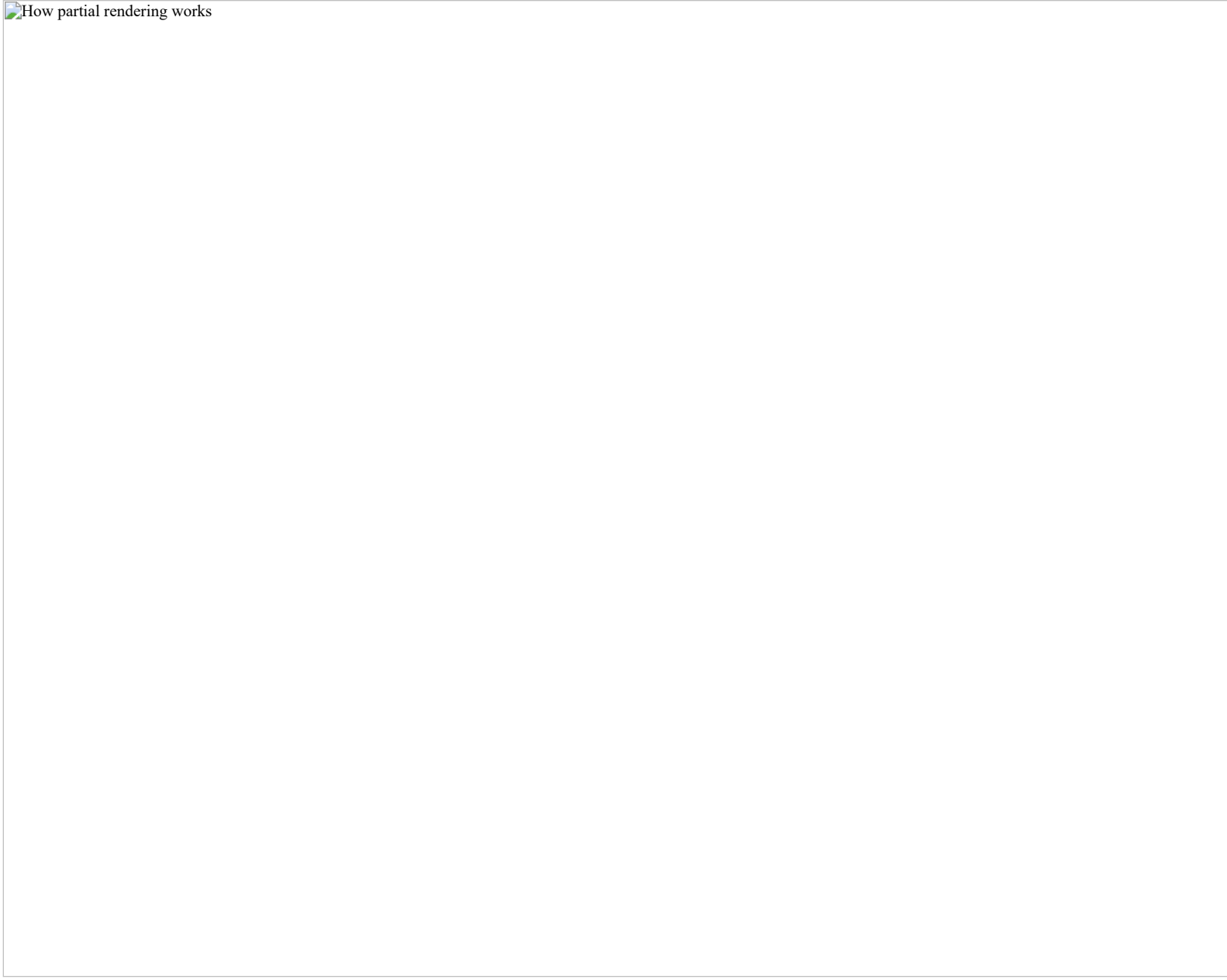
This means on navigation, the cache is reused as much as possible, instead of making a new request to the server - improving performance by reducing the number of requests and data transferred.

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` page will be unmounted, the `analytics` page will be mounted with fresh state, and the shared `dashboard` layout will be preserved. This behavior is also present between two routes on the same dynamic segment e.g. with `/blog/[slug]/page` and navigating from `/blog/first` to `/blog/second`.



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.

7. Routing between `pages/` and `app/`

When incrementally migrating from `pages/` to `app/`, the Next.js router will automatically handle hard navigation between the two. To detect transitions from `pages/` to `app/`, there is a client router filter that leverages probabilistic checking of app routes, which can occasionally result in false positives. By default, such occurrences should be very rare, as we configure the false positive likelihood to be 0.01%. This likelihood can be customized via the `experimental.clientRouterFilterAllowedRate` option in `next.config.js`. It's important to note that lowering the false positive rate will increase the size of the generated filter in the client bundle.

Alternatively, if you prefer to disable this handling completely and manage the routing between `pages/` and `app/` manually, you can set `experimental.clientRouterFilter` to false in `next.config.js`. When this feature is disabled, any dynamic routes in pages that overlap with app routes won't be navigated to properly by default.

---

title: Error Handling description: Learn how to display expected errors and handle uncaught exceptions. related: links: - app/api-reference/file-conventions/error

Errors can be divided into two categories: **expected errors** and **uncaught exceptions**:

- **Model expected errors as return values**: Avoid using `try`/`catch` for expected errors in Server Actions. Use `useFormState` to manage these errors and return them to the client.
- **Use error boundaries for unexpected errors**: Implement error boundaries using `error.tsx` and `global-error.tsx` files to handle unexpected errors and provide a fallback UI.

Handling Expected Errors

Expected errors are those that can occur during the normal operation of the application, such as those from server-side form validation or failed requests. These errors should be handled explicitly and returned to the client.

Handling Expected Errors from Server Actions

Use the `useFormState` hook to manage the state of Server Actions, including handling errors. This approach avoids `try`/`catch` blocks for expected errors, which should be modeled as return values rather than thrown exceptions.

```
'use server'

import { redirect } from 'next/navigation'

export async function createUser(prevState: any, formData: FormData) {
  const res = await fetch('https://...')
  const json = await res.json()

  if (!res.ok) {
    return { message: 'Please enter a valid email' }
  }

  redirect('/dashboard')
}
```

```
'use server'

import { redirect } from 'next/navigation'

export async function createUser(prevState, formData) {
  const res = await fetch('https://...')
  const json = await res.json()

  if (!res.ok) {
    return { message: 'Please enter a valid email' }
  }

  redirect('/dashboard')
}
```

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">{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">{state?.message}</p>
      <button>Sign up</button>
    </form>
  )
}
```

> **Good to know**: These examples use React's `useFormState` hook, which is bundled with the Next.js App Router. If you are using React 19, use `useActionState` instead. See the React docs for more information.

You could also use the returned state to display a toast message from the client component.

Handling Expected Errors from Server Components

When fetching data inside of a Server Component, you can use the response to conditionally render an error message or `redirect`.

```
export default async function Page() {
  const res = await fetch(`https://...`)
  const data = await res.json()

  if (!res.ok) {
    return 'There was an error.'
  }

  return '...'
}
```

```
export default async function Page() {
  const res = await fetch(`https://...`)
  const data = await res.json()

  if (!res.ok) {
    return 'There was an error.'
  }
```

```
  return '...'
}
```

Uncaught Exceptions

Uncaught exceptions are unexpected errors that indicate bugs or issues that should not occur during the normal flow of your application. These should be handled by throwing errors, which will then be caught by error boundaries.

- **Common:** Handle uncaught errors below the root layout with `error.js`.
- **Optional:** Handle granular uncaught errors with nested `error.js` files (e.g. `app/dashboard/error.js`)
- **Uncommon:** Handle uncaught errors in the root layout with `global-error.js`.

Using Error Boundaries

Next.js uses error boundaries to handle uncaught exceptions. Error boundaries catch errors in their child components and display a fallback UI instead of the component tree that crashed.

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

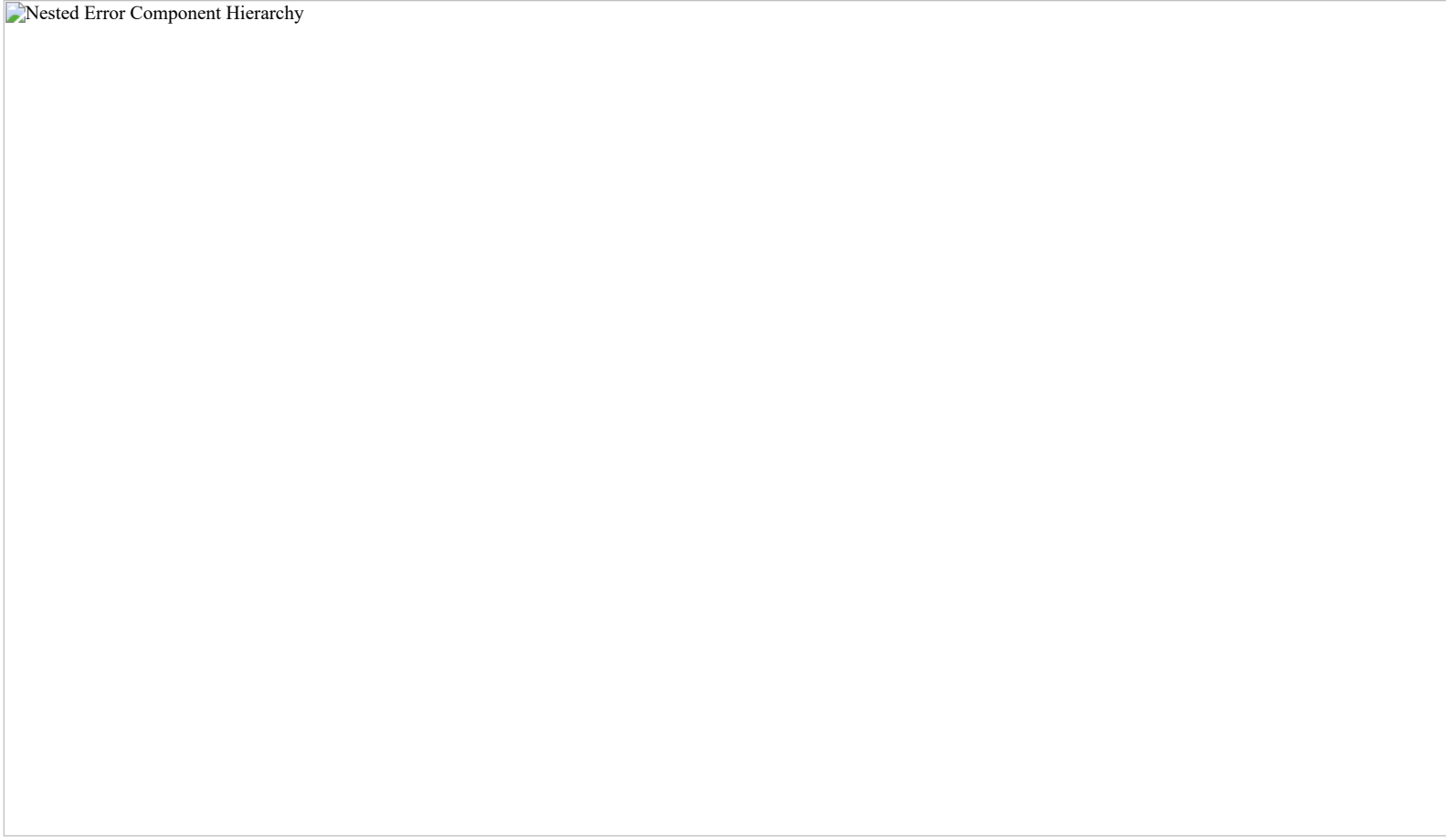
```
'use client' // Error boundaries must be Client Components

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 boundaries 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>
  )
}
```

If you want errors to bubble up to the parent error boundary, you can `throw` when rendering the `error` component.

Handling Errors in Nested Routes

Errors will bubble up to the nearest parent error boundary. This allows for granular error handling by placing `error.tsx` files at different levels in the route hierarchy.



Handling Global Errors

While less common, you can handle errors in the root layout using `app/global-error.js`, located in the root app directory, even when leveraging internationalization. Global error UI must define its own `<html>` and `<body>` tags, since it is replacing the root layout or template when active.

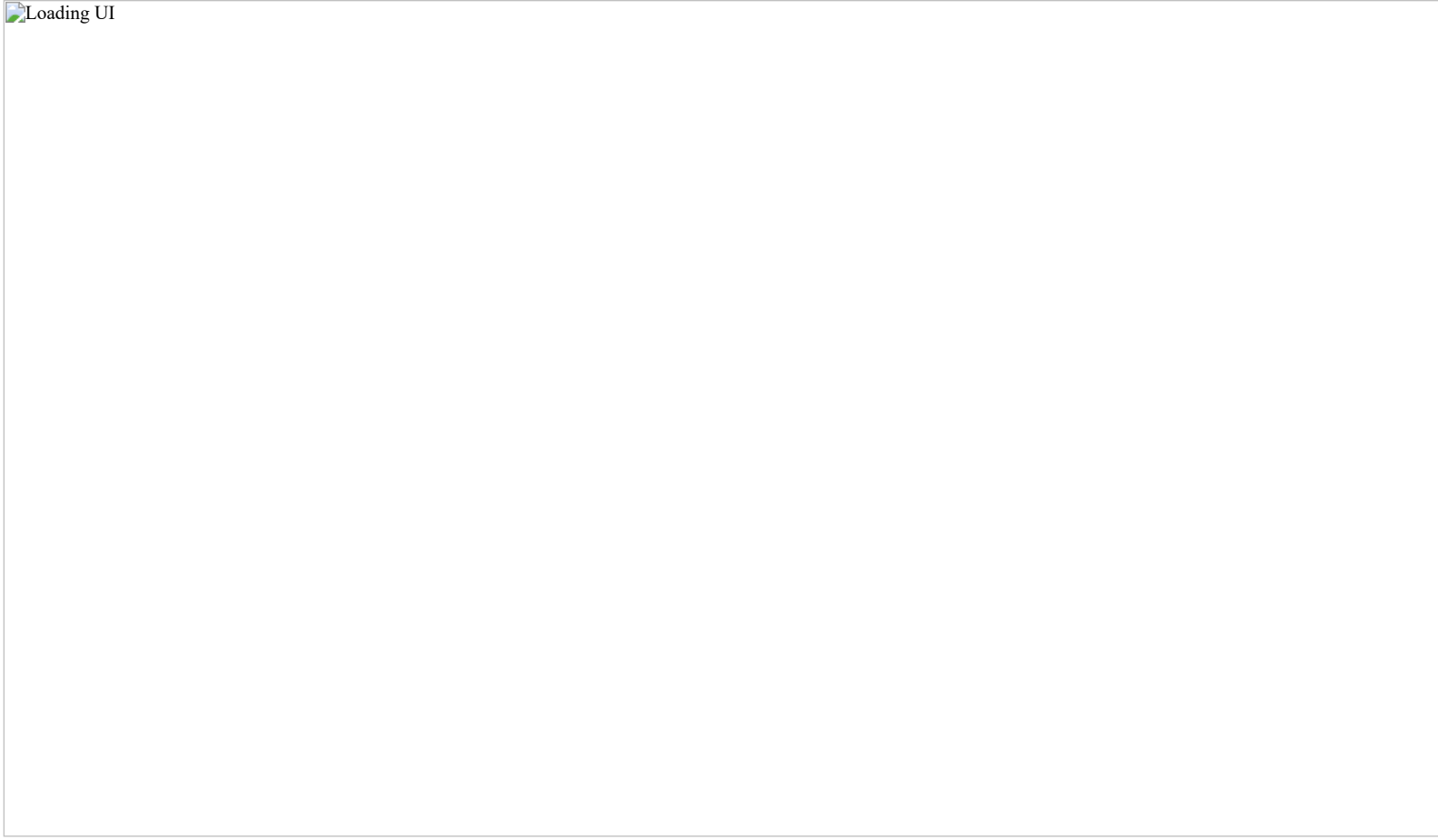
```
'use client' // Error boundaries must be Client Components

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

```
'use client' // Error boundaries must be Client Components

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

---

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.

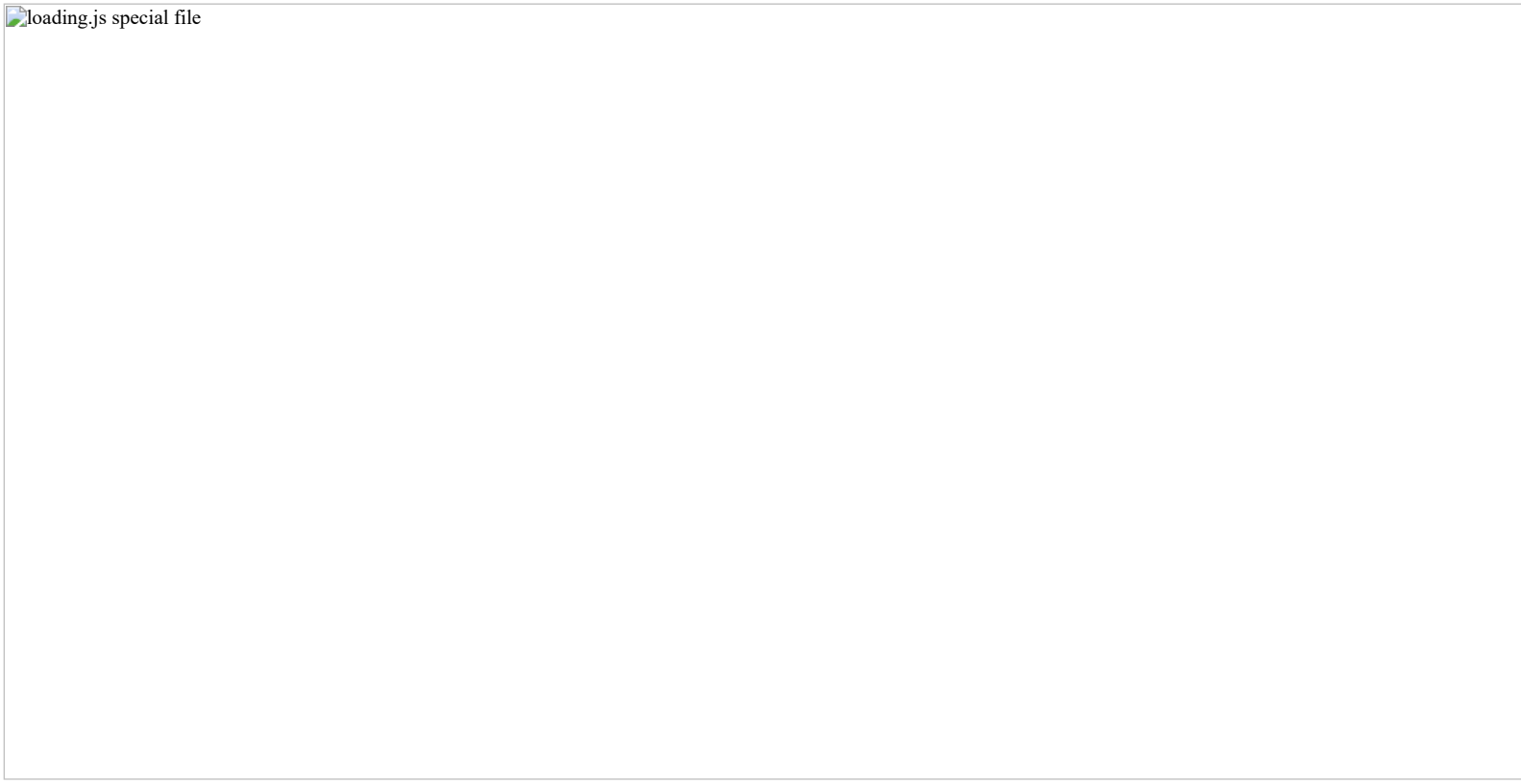
The special file `loading.js` helps you create meaningful Loading UI with React Suspense. With this convention, you can show an instant loading state from the server while the content of a route segment loads. The new content is automatically swapped in once rendering is complete.



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.

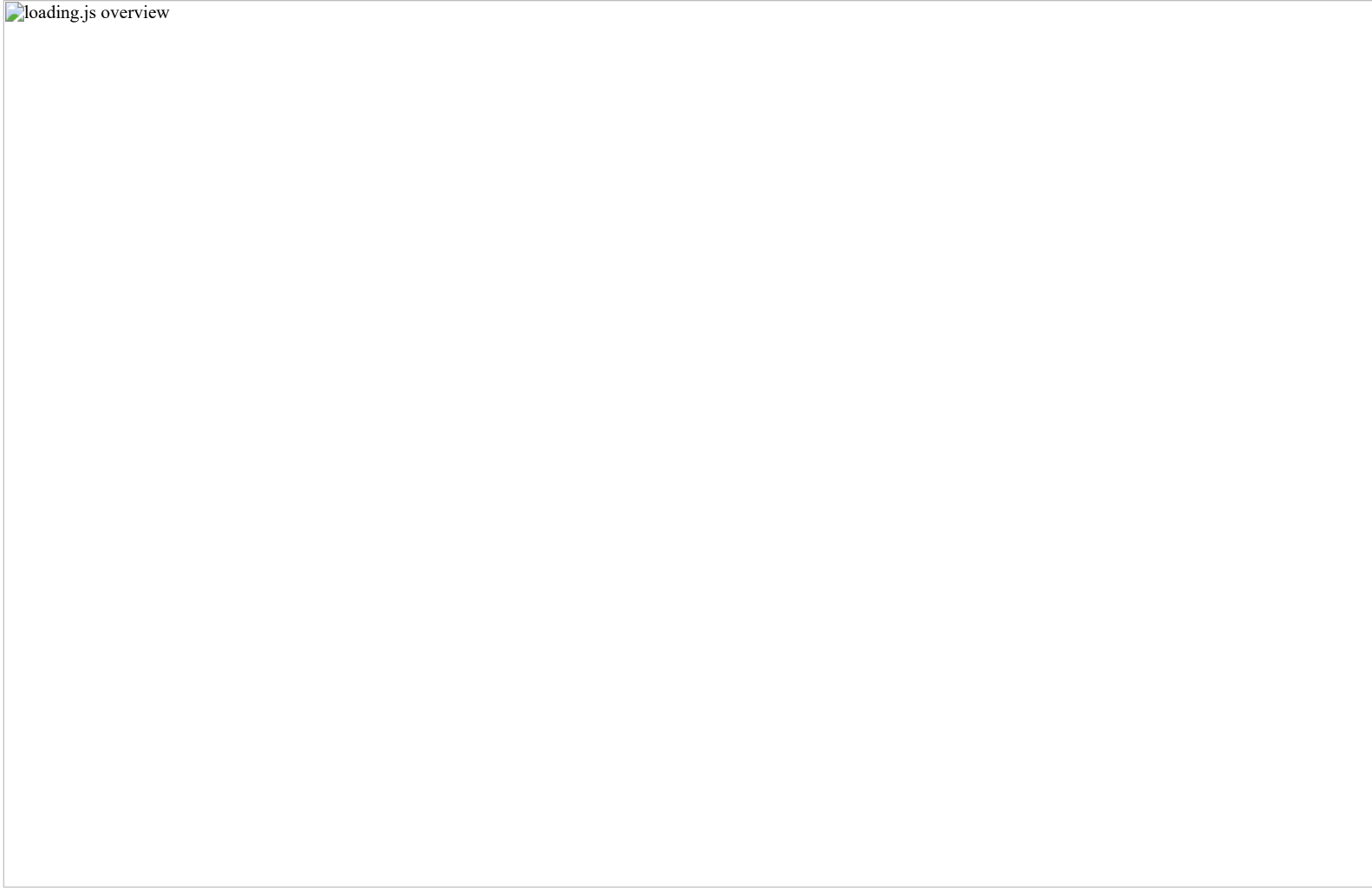
Create a loading state by adding a `loading.js` file inside a folder.




```
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.

Good to know:

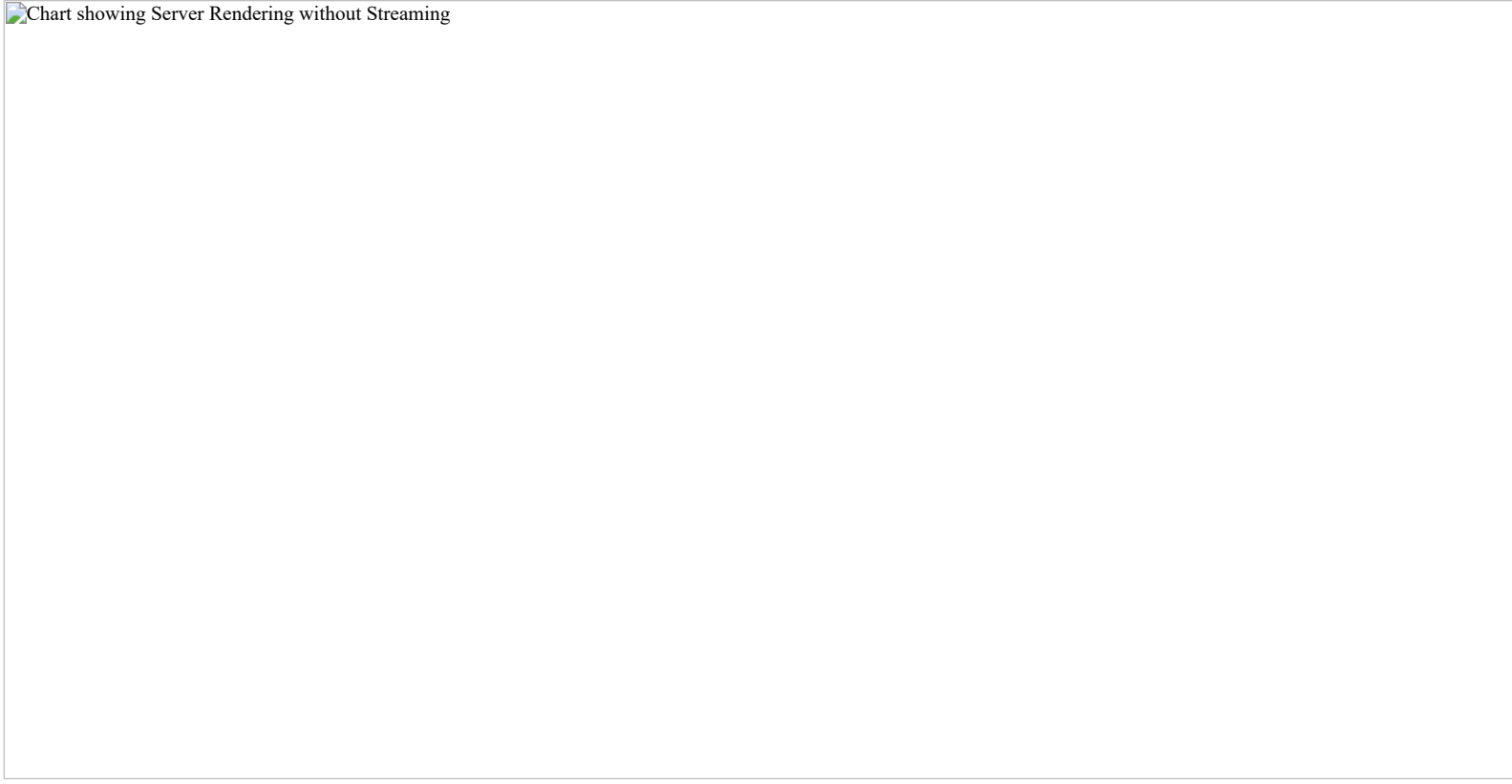
- Some browsers buffer a streaming response. You may not see the streamed response until the response exceeds 1024 bytes. This typically only affects "hello world" applications, but not real applications.

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.

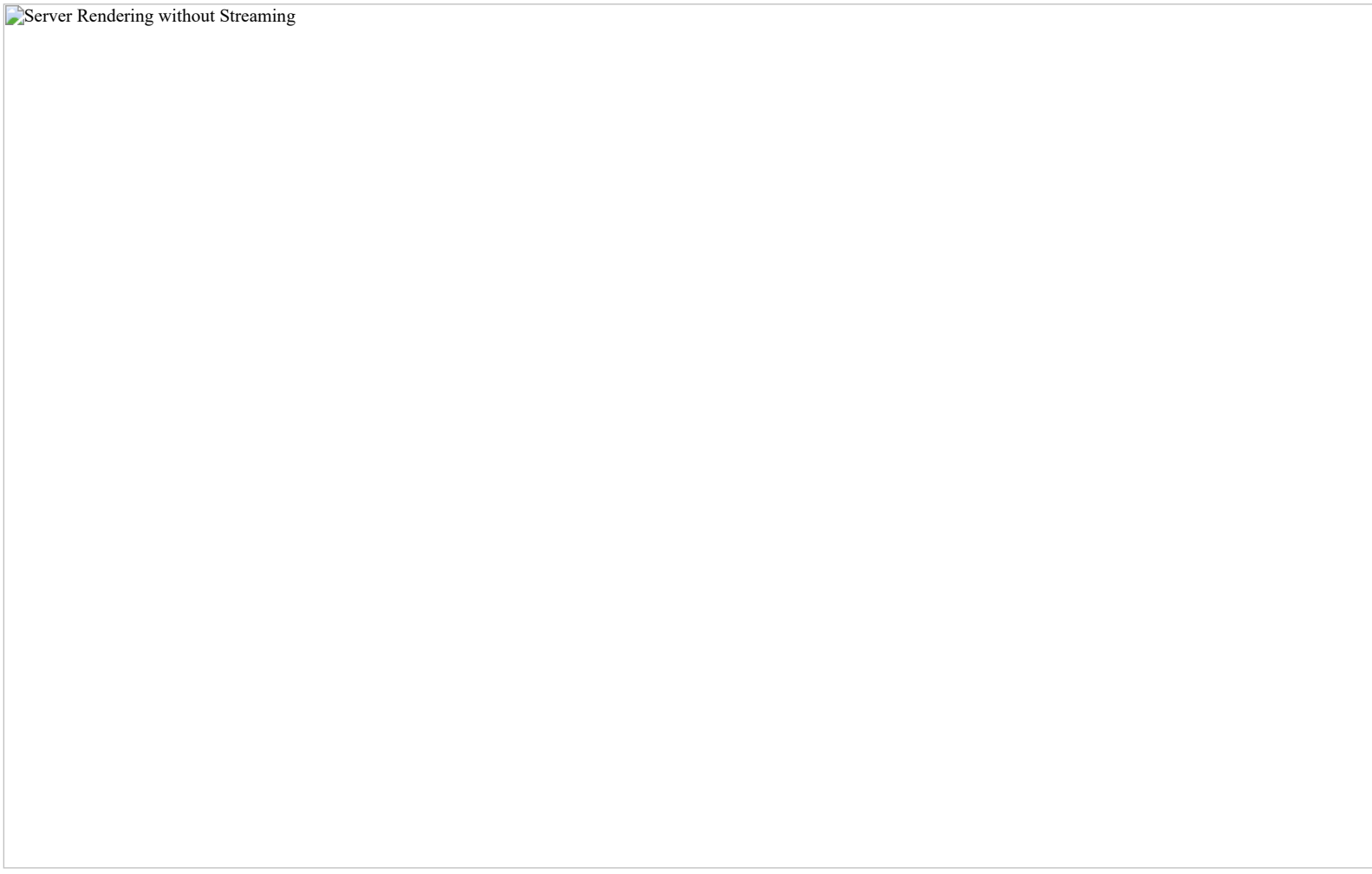
With SSR, there's a series of steps that need to be completed before a user can see and interact with a page:

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 Rich Results 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: Redirecting description: Learn the different ways to handle redirects in Next.js. related: links: - app/api-reference/functions/redirect - app/api-reference/functions/permanentRedirect - app/building-your-application/routing/middleware - app/api-reference/next-config-js/redirects

There are a few ways you can handle redirects in Next.js. This page will go through each available option, use cases, and how to manage large numbers of redirects.

API	Purpose	Where	Status Code
`redirect`	Redirect user after a mutation or event	Server Components, Server Actions, Route Handlers	307 (Temporary) or 303 (Server Action)
`permanentRedirect`	Redirect user after a mutation or event	Server Components, Server Actions, Route Handlers	308 (Permanent)
`useRouter`	Perform a client-side navigation	Event Handlers in Client Components	N/A
`redirects in next.config.js`	Redirect an incoming request based on a path	`next.config.js` file	307 (Temporary) or 308 (Permanent)
`NextResponse.redirect`	Redirect an incoming request based on a condition	Middleware	Any

API	Purpose	Where	Status Code
`useRouter`	Perform a client-side navigation	Components	N/A
`redirects in next.config.js`	Redirect an incoming request based on a path	`next.config.js` file	307 (Temporary) or 308 (Permanent)
`NextResponse.redirect`	Redirect an incoming request based on a condition	Middleware	Any

`redirect` function

The `redirect` function allows you to redirect the user to another URL. You can call `redirect` in Server Components, Route Handlers, and Server Actions.

`redirect` is often used after a mutation or event. For example, creating a post:

```
'use server'

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

export async function createPost(id: string) {
  try {
    // Call database
  } catch (error) {
    // Handle errors
```

```
  }

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

```
'use server'

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

export async function createPost(id) {
  try {
    // Call database
  } catch (error) {
    // Handle errors
  }

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

Good to know:

- `redirect` returns a 307 (Temporary Redirect) status code by default. When used in a Server Action, it returns a 303 (See Other), which is commonly used for redirecting to a success page as a result of a POST request.
- `redirect` internally throws an error so it should be called outside of `try/catch` blocks.
- `redirect` can be called in Client Components during the rendering process but not in event handlers. You can use the `useRouter` hook instead.
- `redirect` also accepts absolute URLs and can be used to redirect to external links.
- If you'd like to redirect before the render process, use `next.config.js` or Middleware.

See the `redirect` API reference for more information.

`permanentRedirect` function

The `permanentRedirect` function allows you to **permanently** redirect the user to another URL. You can call `permanentRedirect` in Server Components, Route Handlers, and Server Actions.

`permanentRedirect` is often used after a mutation or event that changes an entity's canonical URL, such as updating a user's profile URL after they change their username:

```
'use server'

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

export async function updateUsername(username: string, formData: FormData) {
  try {
    // Call database
  } catch (error) {
    // Handle errors
  }

  revalidateTag('username') // Update all references to the username
  permanentRedirect(`/profile/${username}`) // Navigate to the new user profile
}
```

```
'use server'

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

export async function updateUsername(username, formData) {
  try {
    // Call database
  } catch (error) {
    // Handle errors
  }

  revalidateTag('username') // Update all references to the username
  permanentRedirect(`/profile/${username}`) // Navigate to the new user profile
}
```

Good to know:

- `permanentRedirect` returns a 308 (permanent redirect) status code by default.
- `permanentRedirect` also accepts absolute URLs and can be used to redirect to external links.
- If you'd like to redirect before the render process, use `next.config.js` or Middleware.

See the `permanentRedirect` API reference for more information.

`useRouter()` hook

If you need to redirect inside an event handler in a Client Component, you can use the `push` method from the `useRouter` hook. For example:

```
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```

```
'use client'

import { useRouter } from 'next/navigation'

export default function Page() {
  const router = useRouter()
```

```
  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```

If you need to redirect inside a component, you can use the `push` method from the `useRouter` hook. For example:

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

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```


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

export default function Page() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      Dashboard
    </button>
  )
}
```


Good to know:

- If you don't need to programmatically navigate a user, you should use a `<Link>` component.

See the `useRouter` API reference for more information.

See the `useRouter` API reference for more information.

`redirects` in `next.config.js`

The `redirects` option in the `next.config.js` file allows you to redirect an incoming request path to a different destination path. This is useful when you change the URL structure of pages or have a list of redirects that are known ahead of time.

`redirects` supports path, header, cookie, and query matching, giving you the flexibility to redirect users based on an incoming request.

To use `redirects`, add the option to your `next.config.js` file:

```
module.exports = {
  async redirects() {
    return [
      // Basic redirect
      {
        source: '/about',
        destination: '/',
        permanent: true,
      },
      // Wildcard path matching
      {
        source: '/blog/:slug',
        destination: '/news/:slug',
        permanent: true,
      },
    ]
  },
}
```

See the `redirects` API reference for more information.

Good to know:

- `redirects` can return a 307 (Temporary Redirect) or 308 (Permanent Redirect) status code with the `permanent` option.
- `redirects` may have a limit on platforms. For example, on Vercel, there's a limit of 1,024 redirects. To manage a large number of redirects (1000+), consider creating a custom solution using Middleware. See managing redirects at scale for more.
- `redirects` runs **before** Middleware.

`NextResponse.redirect` in Middleware

Middleware allows you to run code before a request is completed. Then, based on the incoming request, redirect to a different URL using `NextResponse.redirect`. This is useful if you want to redirect users based on a condition (e.g. authentication, session management, etc) or have a large number of redirects.

For example, to redirect the user to a `/login` page if they are not authenticated:

```
import { NextResponse, NextRequest } from 'next/server'
import { authenticate } from 'auth-provider'

export function middleware(request: NextRequest) {
  const isAuthenticated = authenticate(request)

  // If the user is authenticated, continue as normal
  if (isAuthenticated) {
    return NextResponse.next()
  }

  // Redirect to login page if not authenticated
  return NextResponse.redirect(new URL('/login', request.url))
}

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

```
import { NextResponse } from 'next/server'
import { authenticate } from 'auth-provider'

export function middleware(request) {
  const isAuthenticated = authenticate(request)

  // If the user is authenticated, continue as normal
  if (isAuthenticated) {
    return NextResponse.next()
  }

  // Redirect to login page if not authenticated
  return NextResponse.redirect(new URL('/login', request.url))
}

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

Good to know:

- Middleware runs **after** `redirects` in `next.config.js` and **before** rendering.

See the Middleware documentation for more information.

Managing redirects at scale (advanced)

To manage a large number of redirects (1000+), you may consider creating a custom solution using Middleware. This allows you to handle redirects programmatically without having to redeploy your application.

To do this, you'll need to consider:

1. Creating and storing a redirect map.
2. Optimizing data lookup performance.

Next.js Example: See our Middleware with Bloom filter example for an implementation of the recommendations below.

1. Creating and storing a redirect map

A redirect map is a list of redirects that you can store in a database (usually a key-value store) or JSON file.

Consider the following data structure:

```
{
  "/old": {
    "destination": "/new",
    "permanent": true
  },
  "/blog/post-old": {
    "destination": "/blog/post-new",
    "permanent": true
  }
}
```

In Middleware, you can read from a database such as Vercel's Edge Config or Redis, and redirect the user based on the incoming request:

```
import { NextResponse, NextRequest } from 'next/server'
import { get } from '@vercel/edge-config'

type RedirectEntry = {
  destination: string
  permanent: boolean
}

export async function middleware(request: NextRequest) {
  const pathname = request.nextUrl.pathname
  const redirectData = await get(pathname)

  if (redirectData && typeof redirectData === 'string') {
    const redirectEntry: RedirectEntry = JSON.parse(redirectData)
    const statusCode = redirectEntry.permanent ? 308 : 307
    return NextResponse.redirect(redirectEntry.destination, statusCode)
  }

  // No redirect found, continue without redirecting
  return NextResponse.next()
}
```

```
import { NextResponse } from 'next/server'
import { get } from '@vercel/edge-config'

export async function middleware(request) {
  const pathname = request.nextUrl.pathname
  const redirectData = await get(pathname)

  if (redirectData) {
    const redirectEntry = JSON.parse(redirectData)
    const statusCode = redirectEntry.permanent ? 308 : 307
    return NextResponse.redirect(redirectEntry.destination, statusCode)
  }

  // No redirect found, continue without redirecting
  return NextResponse.next()
}
```

2. Optimizing data lookup performance

Reading a large dataset for every incoming request can be slow and expensive. There are two ways you can optimize data lookup performance:

- Use a database that is optimized for fast reads, such as Vercel Edge Config or Redis.
- Use a data lookup strategy such as a Bloom filter to efficiently check if a redirect exists **before** reading the larger redirects file or database.

Considering the previous example, you can import a generated bloom filter file into Middleware, then, check if the incoming request pathname exists in the bloom filter.

If it does, forward the request to a Route Handler API Routes which will check the actual file and redirect the user to the appropriate URL. This avoids importing a large redirects file into Middleware, which can slow down every incoming request.

```
import { NextResponse, NextRequest } from 'next/server'
import { ScalableBloomFilter } from 'bloom-filters'
import GeneratedBloomFilter from './redirects/bloom-filter.json'

type RedirectEntry = {
  destination: string
  permanent: boolean
}

// Initialize bloom filter from a generated JSON file
const bloomFilter = ScalableBloomFilter.fromJSON(GeneratedBloomFilter as any)

export async function middleware(request: NextRequest) {
  // Get the path for the incoming request
  const pathname = request.nextUrl.pathname

  // Check if the path is in the bloom filter
  if (bloomFilter.has(pathname)) {
    // Forward the pathname to the Route Handler
    const api = new URL(
      `/api/redirects?pathname=${encodeURIComponent(request.nextUrl.pathname)}`,
      request.nextUrl.origin
    )

    try {
      // Fetch redirect data from the Route Handler
      const redirectData = await fetch(api)

      if (redirectData.ok) {
        const redirectEntry: RedirectEntry | undefined =
          await redirectData.json()

        if (redirectEntry) {
          // Determine the status code
          const statusCode = redirectEntry.permanent ? 308 : 307

          // Redirect to the destination
          return NextResponse.redirect(redirectEntry.destination, statusCode)
        }
      }
    } catch (error) {
      console.error(error)
    }
  }

  // No redirect found, continue the request without redirecting
  return NextResponse.next()
}
```

```
import { NextResponse } from 'next/server'
import { ScalableBloomFilter } from 'bloom-filters'
import GeneratedBloomFilter from './redirects/bloom-filter.json'

// Initialize bloom filter from a generated JSON file
const bloomFilter = ScalableBloomFilter.fromJSON(GeneratedBloomFilter)

export async function middleware(request) {
  // Get the path for the incoming request
  const pathname = request.nextUrl.pathname

  // Check if the path is in the bloom filter
  if (bloomFilter.has(pathname)) {
    // Forward the pathname to the Route Handler
    const api = new URL(
      `/api/redirects?pathname=${encodeURIComponent(request.nextUrl.pathname)}`,
      request.nextUrl.origin
    )

    try {
      // Fetch redirect data from the Route Handler
      const redirectData = await fetch(api)

      if (redirectData.ok) {
        const redirectEntry = await redirectData.json()

        if (redirectEntry) {
          // Determine the status code
          const statusCode = redirectEntry.permanent ? 308 : 307

          // Redirect to the destination
          return NextResponse.redirect(redirectEntry.destination, statusCode)
        }
      }
    } catch (error) {
      console.error(error)
    }
  }

  // No redirect found, continue the request without redirecting
  return NextResponse.next()
}
```

Then, in the Route Handler:

```
import { NextRequest, NextResponse } from 'next/server'
import redirects from '@/app/redirects/redirects.json'

type RedirectEntry = {
  destination: string
  permanent: boolean
}

export function GET(request: NextRequest) {
  const pathname = request.nextUrl.searchParams.get('pathname')
  if (!pathname) {
    return new Response('Bad Request', { status: 400 })
  }
```

```
  // Get the redirect entry from the redirects.json file
  const redirect = (redirects as Record<string, RedirectEntry>)[pathname]

  // Account for bloom filter false positives
  if (!redirect) {
    return new Response('No redirect', { status: 400 })
  }

  // Return the redirect entry
  return NextResponse.json(redirect)
}
```

```
import { NextResponse } from 'next/server'
import redirects from '@/app/redirects/redirects.json'

export function GET(request) {
  const pathname = request.nextUrl.searchParams.get('pathname')
  if (!pathname) {
    return new Response('Bad Request', { status: 400 })
  }

  // Get the redirect entry from the redirects.json file
  const redirect = redirects[pathname]

  // Account for bloom filter false positives
  if (!redirect) {
    return new Response('No redirect', { status: 400 })
  }

  // Return the redirect entry
  return NextResponse.json(redirect)
}
```

Then, in the API Route:

```
import type { NextApiRequest, NextApiResponse } from 'next'
import redirects from '@/app/redirects/redirects.json'

type RedirectEntry = {
  destination: string
  permanent: boolean
}

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  const pathname = req.query.pathname
  if (!pathname) {
    return res.status(400).json({ message: 'Bad Request' })
  }

  // Get the redirect entry from the redirects.json file
  const redirect = (redirects as Record<string, RedirectEntry>)[pathname]

  // Account for bloom filter false positives
  if (!redirect) {
    return res.status(400).json({ message: 'No redirect' })
  }

  // Return the redirect entry
  return res.json(redirect)
}
```

```
import redirects from '@/app/redirects/redirects.json'

export default function handler(req, res) {
  const pathname = req.query.pathname
  if (!pathname) {
    return res.status(400).json({ message: 'Bad Request' })
  }

  // Get the redirect entry from the redirects.json file
  const redirect = redirects[pathname]

  // Account for bloom filter false positives
  if (!redirect) {
    return res.status(400).json({ message: 'No redirect' })
  }

  // Return the redirect entry
  return res.json(redirect)
}
```

Good to know:

- To generate a bloom filter, you can use a library like `bloom-filters`.
- You should validate requests made to your Route Handler to prevent malicious requests.

---

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
- Adding a loading skeleton to specific route in a common segment

Convention

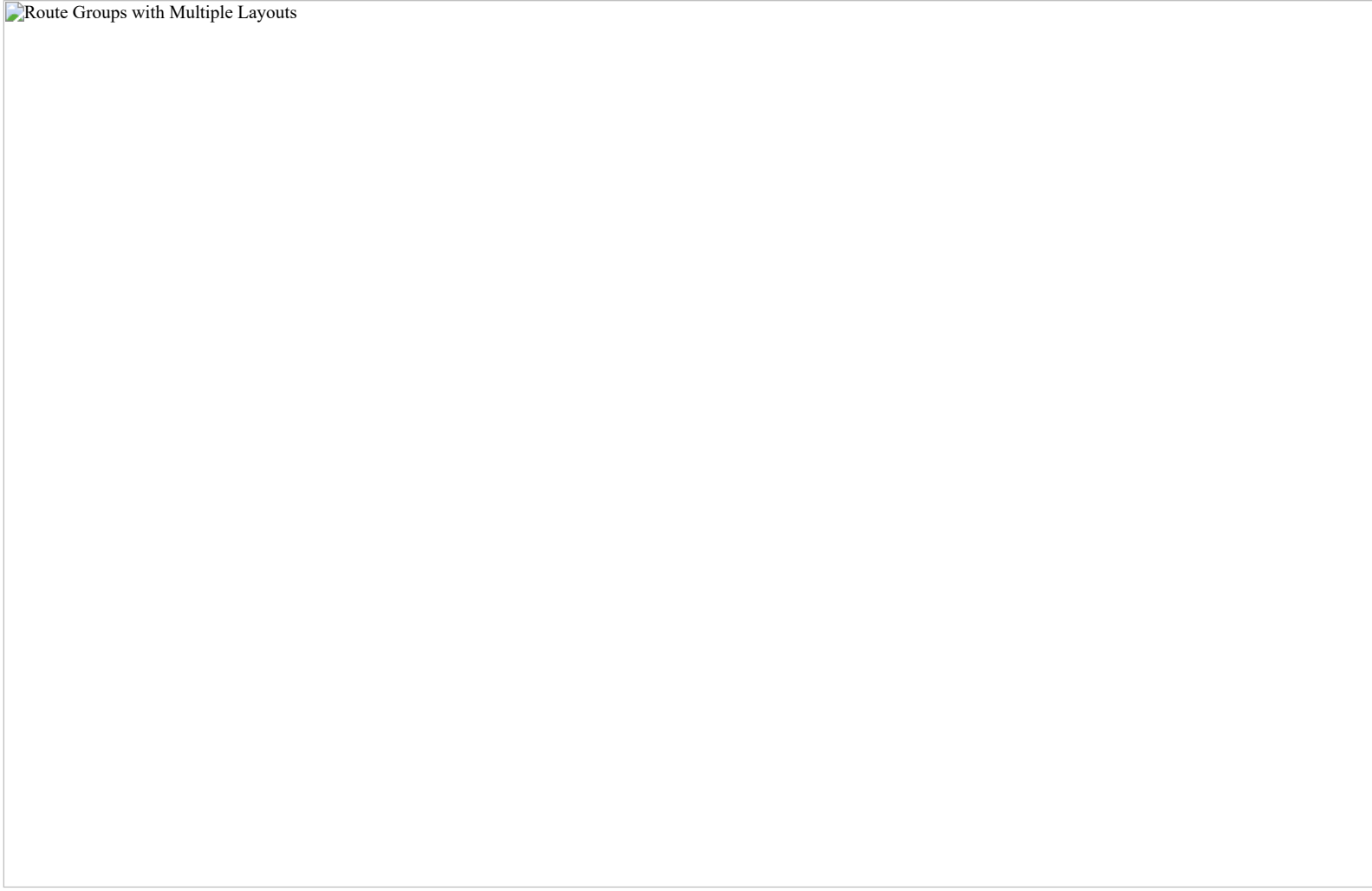
A route group can be created by wrapping a folder's name in parenthesis: `(folderName)`

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.

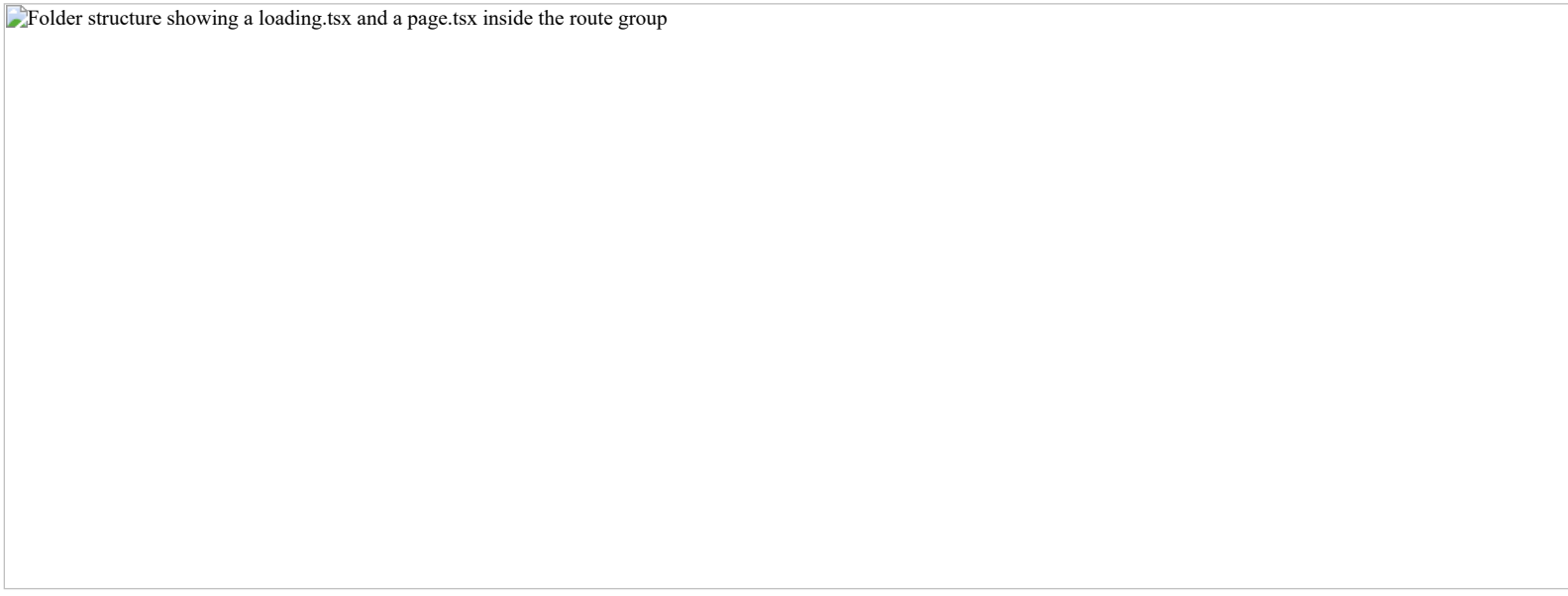


Opting specific segments into a layout

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`).

Opting for loading skeletons on a specific route

To apply a loading skeleton via a `loading.js` file to a specific route, create a new route group (e.g., `/(overview)`) and then move your `loading.tsx` inside that route group.



Now, the `loading.tsx` file will only apply to your dashboard → overview page instead of all your dashboard pages without affecting the URL path structure.

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 group. 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: 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.

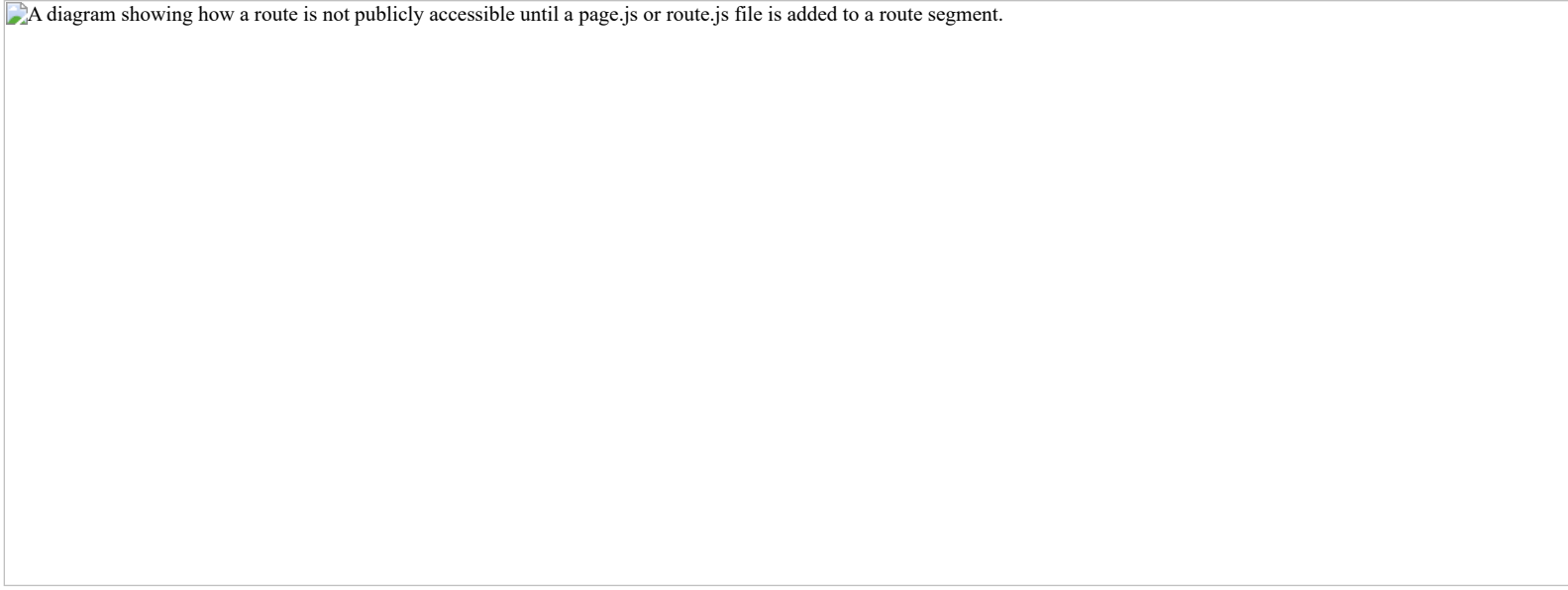
This page shares default behavior and features you can use to organize your project.

Safe colocation by default

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: 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 async function Page({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  const slug = (await params).slug
  return <div>My Post: {slug}</div>
}
```

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

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

Route	Example URL	params
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

- Since the `params` prop is a promise. You must use async/await or React's use function to access the values.
  - In version 14 and earlier, `params` was a synchronous prop. To help with backwards compatibility, you can still access it synchronously in Next.js 15, but this behavior will be deprecated in the future.
- 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	{ slug: undefined }
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 async function Page({
  params,
}: {
  params: Promise<{ slug: string }>
}) {
  return <h1>My Page</h1>
}
```


```
export default async 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: Parallel Routes description: Simultaneously render one or more pages in the same view that can be navigated independently. A pattern for highly dynamic applications. related: links: - app/api-reference/file-

conventions/default

Parallel Routes allows you to simultaneously or conditionally render one or more pages within the same layout. They are useful for highly dynamic sections of an app, such as dashboards and feeds on social sites.

For example, considering a dashboard, you can use parallel routes to simultaneously render the `team` and `analytics` pages:

Slots

Parallel routes are created using named **slots**. Slots are defined with the `@folder` convention. For example, the following file structure defines two slots: `@analytics` and `@team`:

Slots are passed as props to the shared parent layout. For the example above, 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({
  children,
  team,
  analytics,
}: {
  children: React.ReactNode
  analytics: React.ReactNode
  team: React.ReactNode
}) {
  return (
    <>
      {children}
      {team}
      {analytics}
    </>
  )
}
```

```
export default function Layout({ children, team, analytics }) {
  return (
    <>
      {children}
      {team}
      {analytics}
    </>
  )
}
```

However, slots are **not** route segments and do not affect the URL structure. For example, for `/@analytics/views`, the URL will be `/views` since `@analytics` is a slot. Slots are combined with the regular Page component to form the final page associated with the route segment. Because of this, you cannot have separate static and dynamic slots at the same route segment level. If one slot is dynamic, all slots at that level must be dynamic.

> **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`.

Active state and navigation

By default, Next.js keeps track of the active *state* (or subpage) for each slot. However, the content rendered within a slot will depend on the type of navigation:

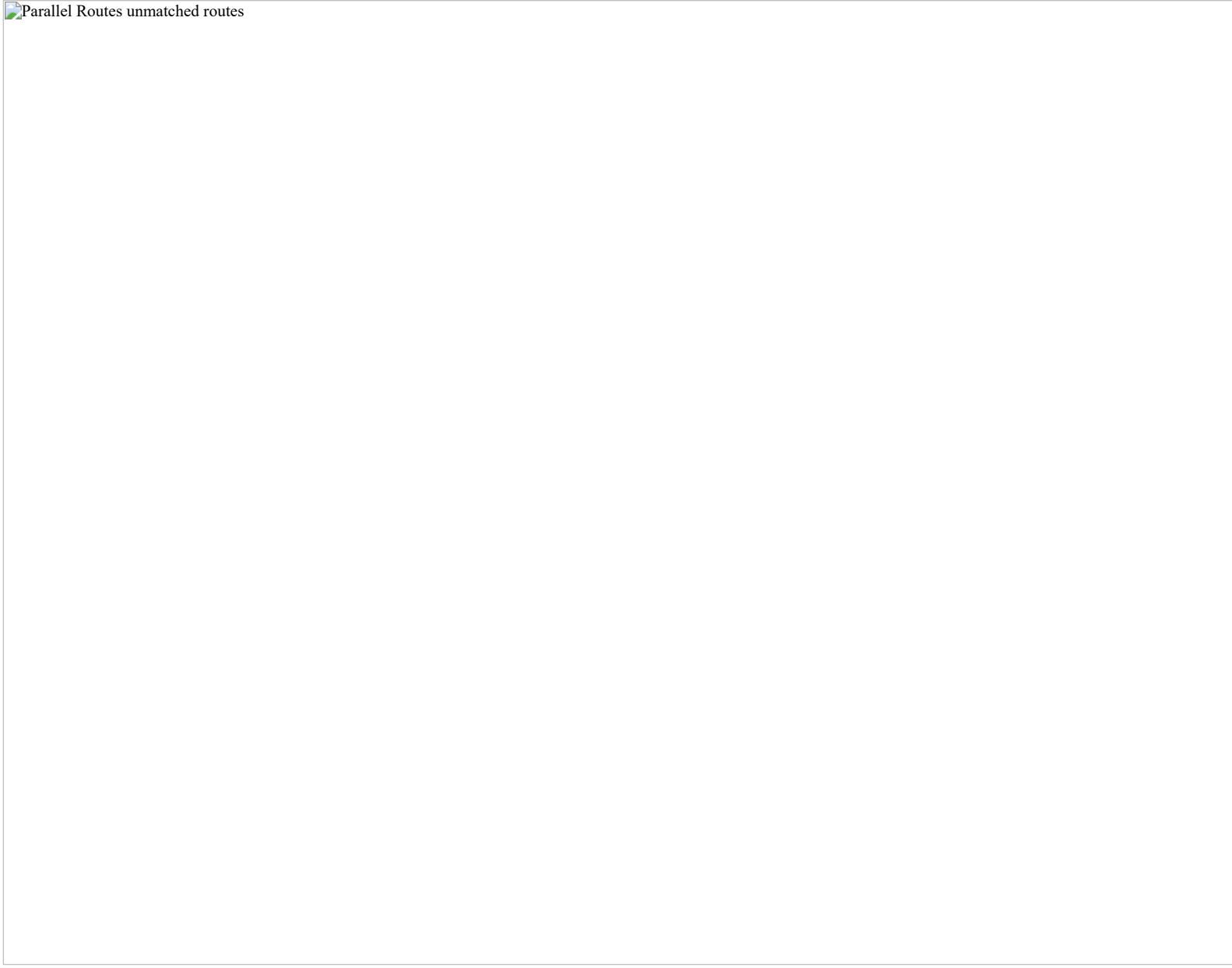
- **Soft Navigation**: During client-side navigation, Next.js will perform a partial render, changing the subpage within the slot, while maintaining the other slot's active subpages, even if they don't match the current URL.
- **Hard Navigation**: After a full-page load (browser refresh), Next.js cannot determine the active state for the slots that don't match the current URL. Instead, it will render a `default.js` file for the unmatched slots, or `404` if `default.js` doesn't exist.

> **Good to know**:
>
> - The `404` for unmatched routes helps ensure that you don't accidentally render a parallel route on a page that it was not intended for.

`default.js`

You can define a `default.js` file to render as a fallback for unmatched slots during the initial load or full-page reload.

Consider the following folder structure. The `@team` slot has a `/settings` page, but `@analytics` does not.



When navigating to `/settings`, the `@team` slot will render the `/settings` page while maintaining the currently active page for the `@analytics` slot.

On refresh, Next.js will render a `default.js` for `@analytics`. If `default.js` doesn't exist, a `404` is rendered instead.

Additionally, since `children` is an implicit slot, you also need to create a `default.js` file to render a fallback for `children` when Next.js cannot recover the active state of the parent page.

`useSelectedLayoutSegment(s)`

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

```
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

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

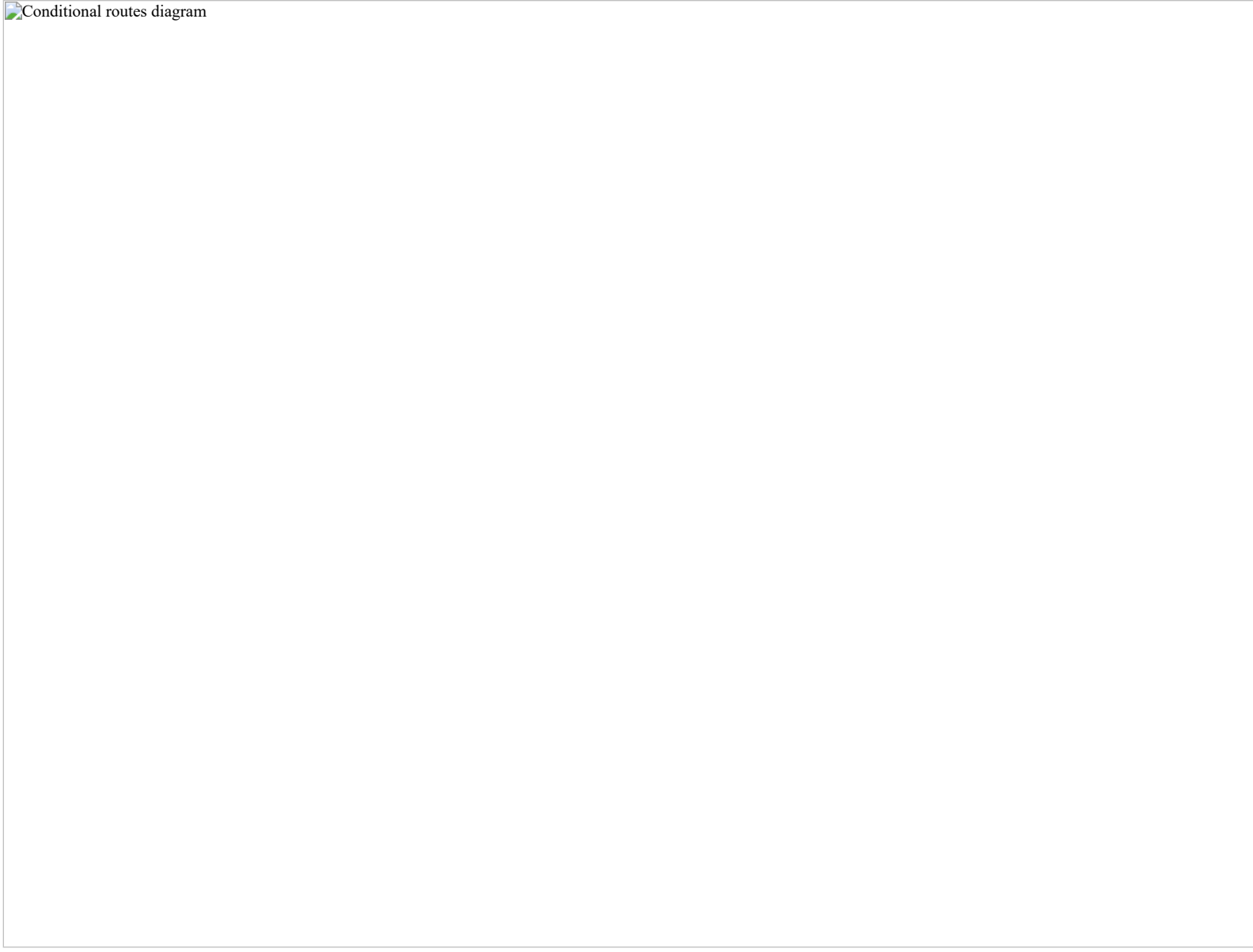
```
'use client'

import { useSelectedLayoutSegment } from 'next/navigation'

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

When a user navigates to `app/@auth/login` (or `/login` in the URL bar), `loginSegment` will be equal to the string `"login"`.

Examples

Conditional Routes

You can use Parallel Routes to conditionally render routes based on certain conditions, such as user role. For example, to render a different dashboard page for the `/admin` or `/user` roles:




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

export default function Layout({
  user,
  admin,
}: {
  user: React.ReactNode
  admin: React.ReactNode
}) {
  const role = checkUserRole()
  return <>{role === 'admin' ? admin : user}</>
}
```

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

export default function Layout({ user, admin }) {
  const role = checkUserRole()
  return <>{role === 'admin' ? admin : user}</>
}
```

Tab Groups

You can add a `layout` inside a slot to allow users to navigate the slot independently. This is useful for creating tabs.

For example, the `@analytics` slot has two subpages: `/page-views` and `/visitors`.

Within `@analytics`, create a `layout` file to share the tabs between the two pages:

```
import Link from 'next/link'

export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <>
      <nav>
        <Link href="/page-views">Page Views</Link>
        <Link href="/visitors">Visitors</Link>
      </nav>
      <div>{children}</div>
    </>
  )
}
```

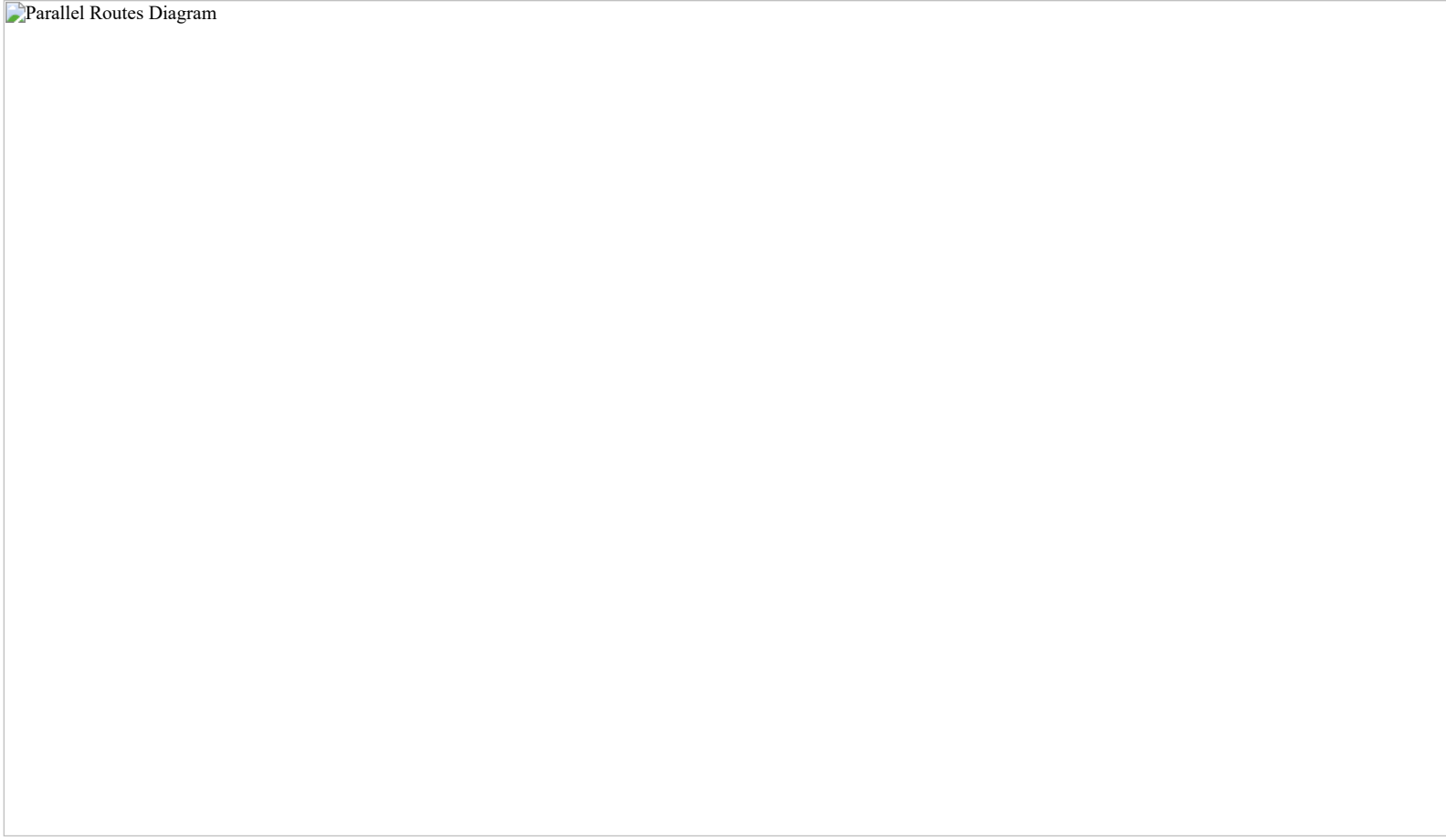
```
import Link from 'next/link'

export default function Layout({ children }) {
  return (
    <>
      <nav>
        <Link href="/page-views">Page Views</Link>
        <Link href="/visitors">Visitors</Link>
      </nav>
      <div>{children}</div>
    </>
  )
}
```

Modals

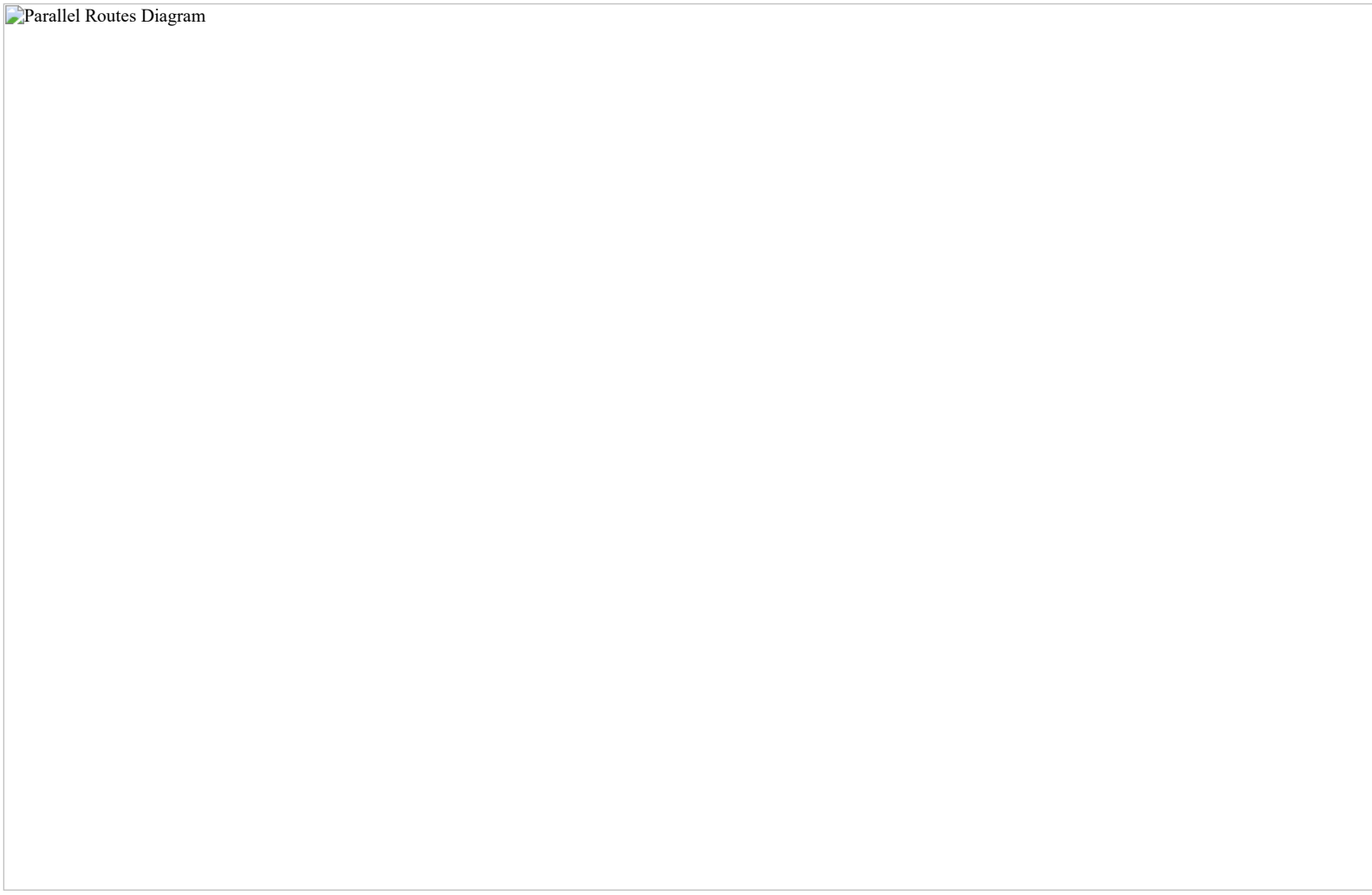
Parallel Routes can be used together with Intercepting Routes to create modals that support deep linking. This allows you to solve common challenges when building modals, such as:

- Making the modal content **shareable through a URL**.
- **Preserving context** when the page is refreshed, instead of closing the modal.
- **Closing the modal on backwards navigation** rather than going to the previous route.
- **Reopening the modal on forwards navigation**.

Consider the following UI pattern, where a user can open a login modal from a layout using client-side navigation, or access a separate `/login` page:



To implement this pattern, start by creating a `/login` route that renders your **main** login page.




```
import { Login } from '@/app/ui/login'

export default function Page() {
  return <Login />
}
```

```
import { Login } from '@/app/ui/login'

export default function Page() {
  return <Login />
}
```

Then, inside the `@auth` slot, add `default.js` file that returns `null`. This ensures that the modal is not rendered when it's not active.

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

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

Inside your `@auth` slot, intercept the `/login` route by updating the `/(.)login` folder. Import the `<Modal>` component and its children into the `/(.)login/page.tsx` file:

```
import { Modal } from '@/app/ui/modal'
import { Login } from '@/app/ui/login'

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

```
import { Modal } from '@/app/ui/modal'
import { Login } from '@/app/ui/login'

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

Good to know:

- The convention used to intercept the route, e.g. `(.)`, depends on your file-system structure. See Intercepting Routes convention.
- By separating the `<Modal>` functionality from the modal content (`<Login>`), you can ensure any content inside the modal, e.g. forms, are Server Components. See Interleaving Client and Server Components for more information.

Opening the modal

Now, you can leverage the Next.js router to open and close the modal. This ensures the URL is correctly updated when the modal is open, and when navigating backwards and forwards.

To open the modal, pass the `@auth` slot as a prop to the parent layout and render it alongside the `children` prop.

```
import Link from 'next/link'

export default function Layout({
  auth,
  children,
}: {
  auth: React.ReactNode
  children: React.ReactNode
}) {
  return (
    <>
      <nav>
        <Link href="/login">Open modal</Link>
      </nav>
      <div>{auth}</div>
      <div>{children}</div>
    </>
  )
}
```

```
import Link from 'next/link'

export default function Layout({ auth, children }) {
  return (
    <>
      <nav>
        <Link href="/login">Open modal</Link>
      </nav>
      <div>{auth}</div>
      <div>{children}</div>
    </>
  )
}
```

When the user clicks the `<Link>`, the modal will open instead of navigating to the `/login` page. However, on refresh or initial load, navigating to `/login` will take the user to the main login page.

Closing the modal

You can close the modal by calling `router.back()` or by using the `Link` component.

```
'use client'

import { useRouter } from 'next/navigation'

export function Modal({ children }: { children: React.ReactNode }) {
  const router = useRouter()

  return (
    <>
      <button
        onClick={() => {
          router.back()
        }}
      >
        Close modal
      </button>
      <div>{children}</div>
    </>
```

```
  )
}
```

```
'use client'

import { useRouter } from 'next/navigation'

export function Modal({ children }) {
  const router = useRouter()

  return (
    <>
      <button
        onClick={() => {
          router.back()
        }}
      >
        Close modal
      </button>
      <div>{children}</div>
    </>
  )
}
```

When using the `Link` component to navigate away from a page that shouldn't render the `@auth` slot anymore, we need to make sure the parallel route matches to a component that returns `null`. For example, when navigating back to the root page, we create a `@auth/page.tsx` component:

```
import Link from 'next/link'

export function Modal({ children }: { children: React.ReactNode }) {
  return (
    <>
      <Link href="/">Close modal</Link>
      <div>{children}</div>
    </>
  )
}
```

```
import Link from 'next/link'

export function Modal({ children }) {
  return (
    <>
      <Link href="/">Close modal</Link>
      <div>{children}</div>
    </>
  )
}

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

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

Or if navigating to any other page (such as `/foo`, `/foo/bar`, etc), you can use a catch-all slot:

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

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

Good to know:

- We use a catch-all route in our `@auth` slot to close the modal because of the behavior described in Active state and navigation. Since client-side navigations to a route that no longer match the slot will remain visible, we need to match the slot to a route that returns `null` to close the modal.
- Other examples could include opening a photo modal in a gallery while also having a dedicated `/photo/[id]` page, or opening a shopping cart in a side modal.
- View an example of modals with Intercepted and Parallel Routes.

Loading and Error UI

Parallel Routes can be streamed independently, allowing you to define independent error and loading states for each route:

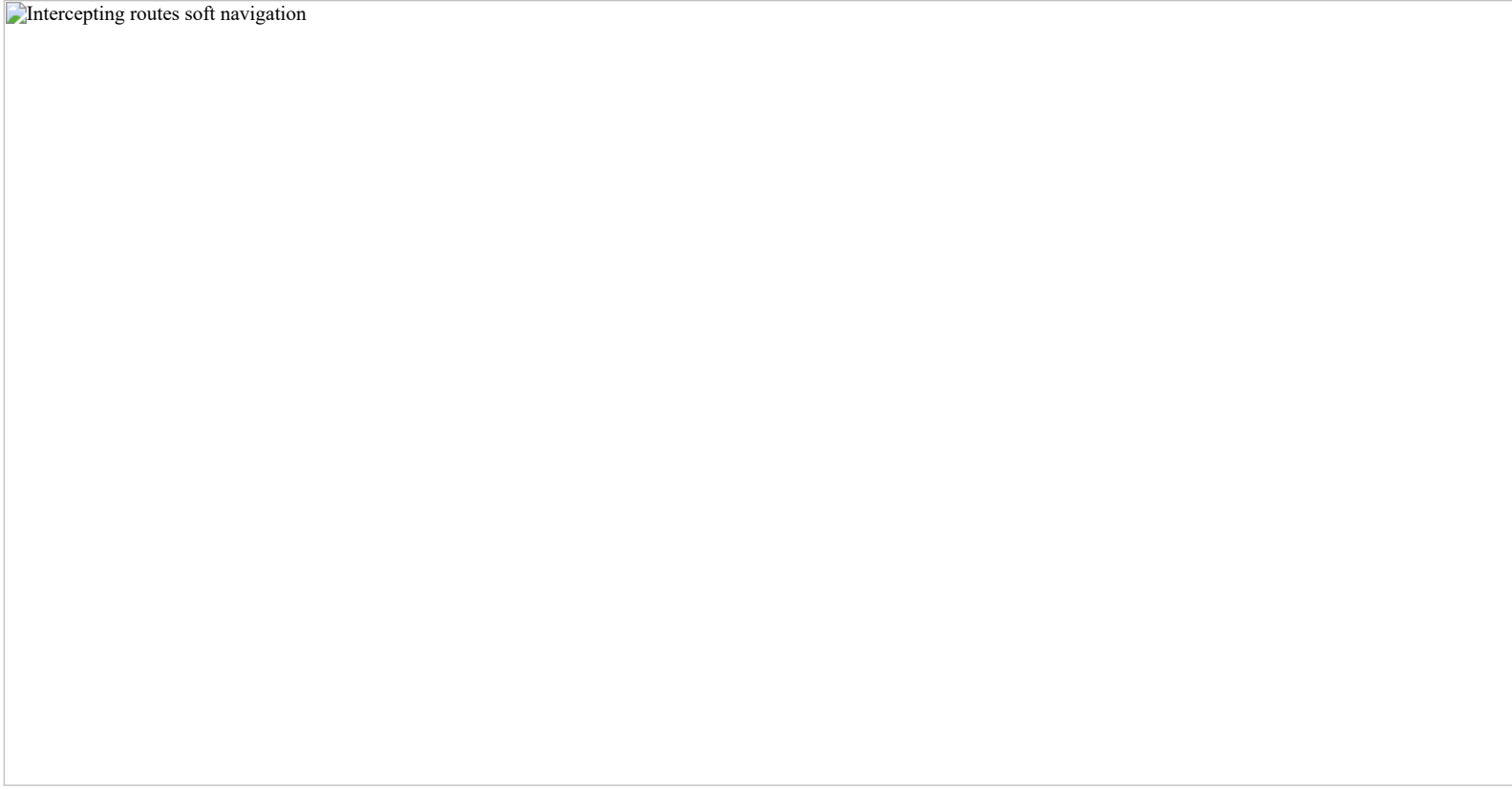
See the Loading UI and Error Handling documentation for more information.

---

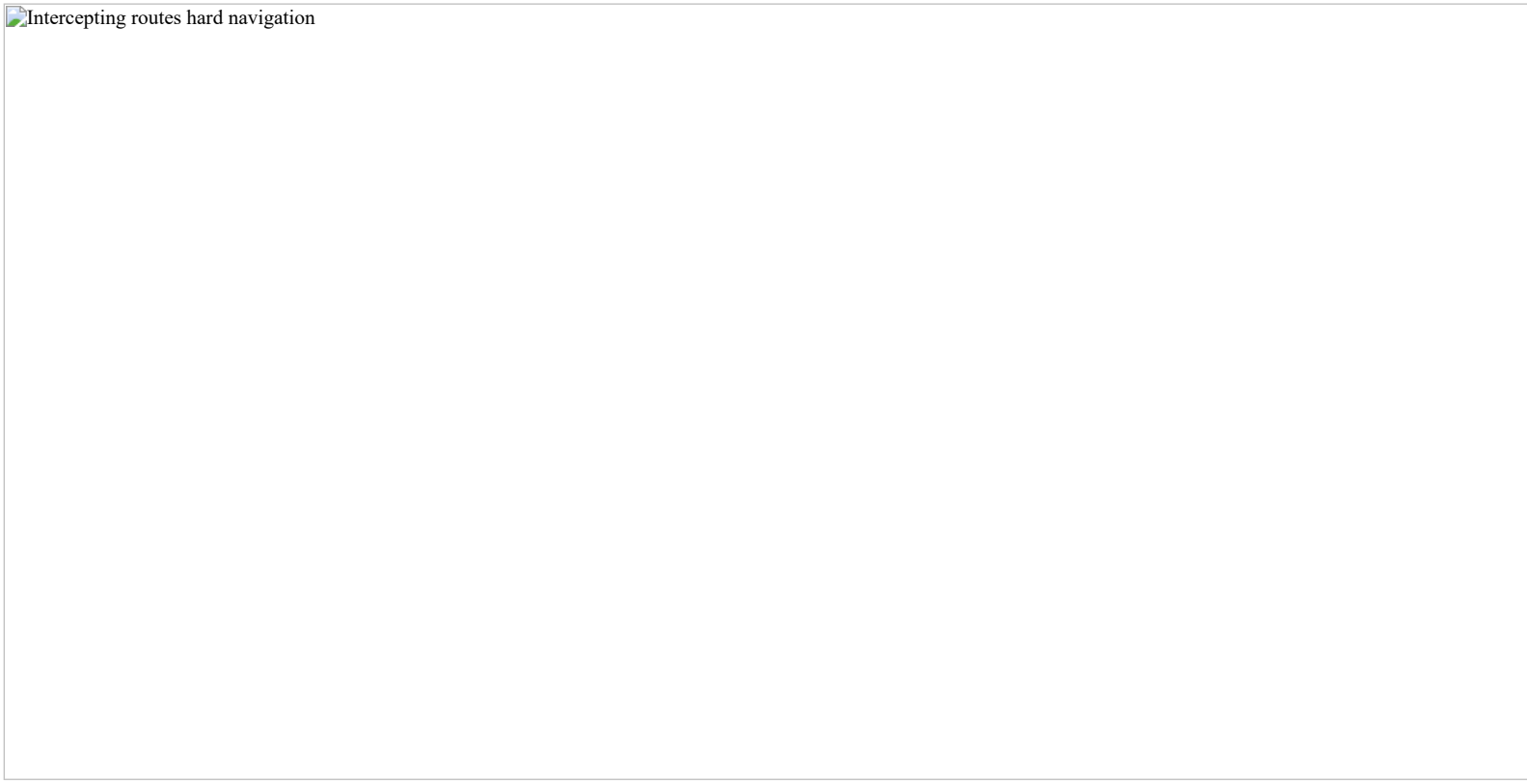
title: Intercepting Routes description: Use intercepting routes to load a new route within the current layout while masking the browser URL, useful for advanced routing patterns such as modals. related: title: Next Steps description: Learn how to use modals with Intercepted and Parallel Routes. links: - 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.



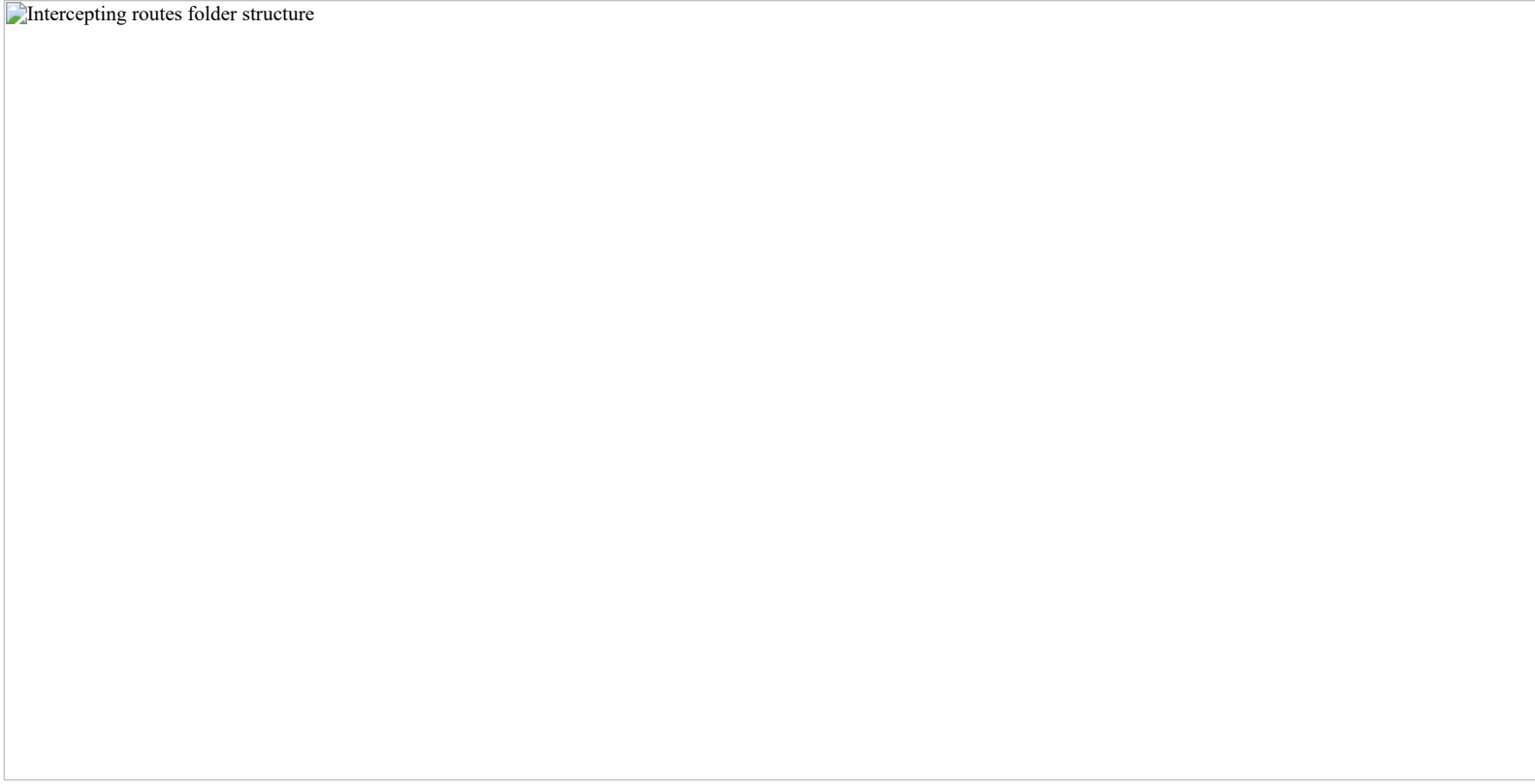
Convention

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.

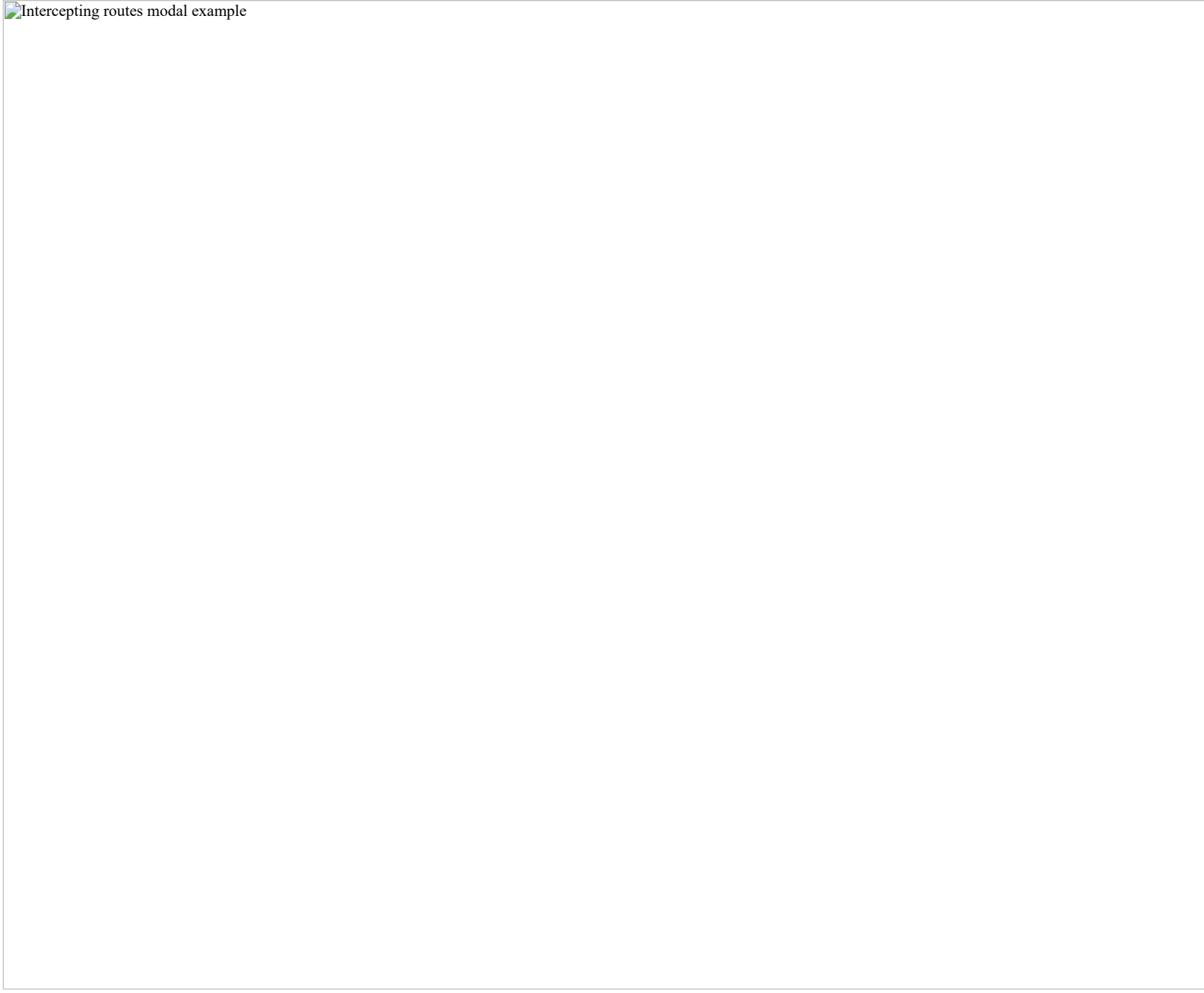
Examples

Modals

Intercepting Routes can be used together with Parallel Routes to create modals. This allows you to solve common challenges when building modals, such as:

- Making the modal content **shareable through a URL**.
- **Preserving context** when the page is refreshed, instead of closing the modal.
- **Closing the modal on backwards navigation** rather than going to the previous route.
- **Reopening the modal on forwards navigation**.

Consider the following UI pattern, where a user can open a photo modal from a gallery using client-side navigation, or navigate to the photo page directly from a shareable URL:



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.

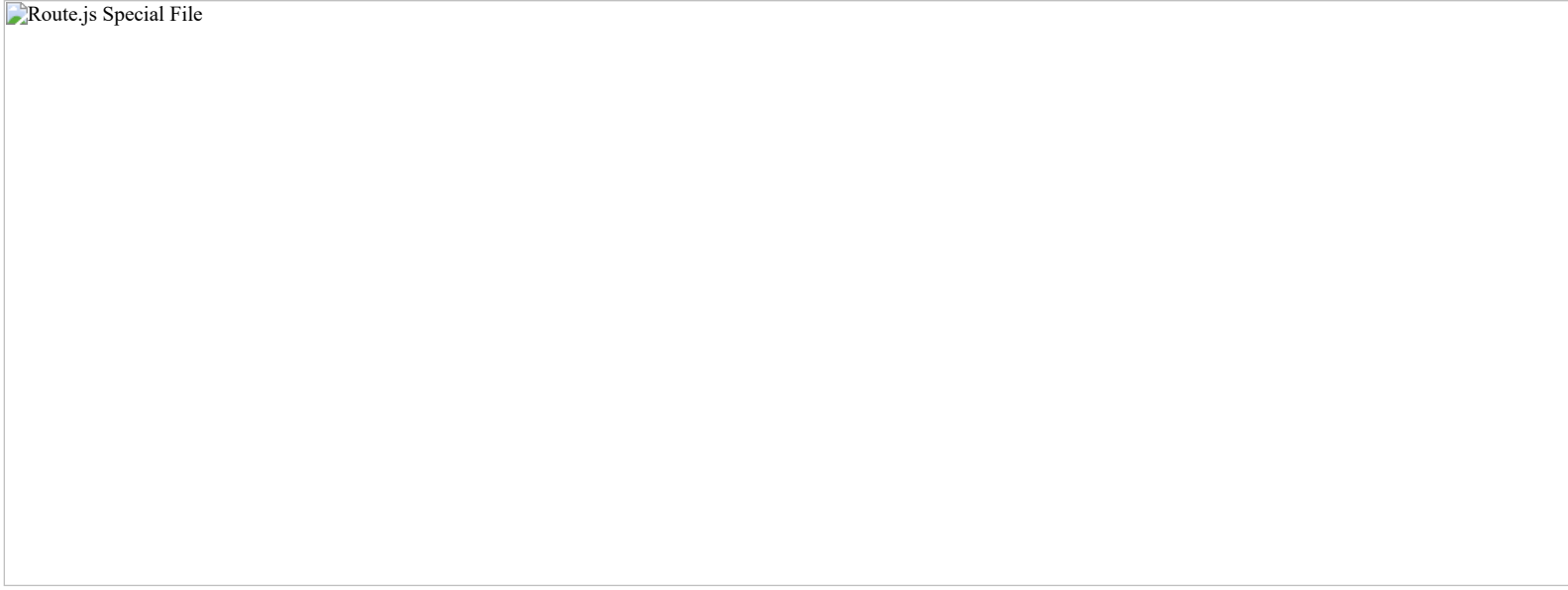
See the Parallel Routes documentation for a step-by-step example, or see our image gallery example.

> **Good to know:**
>
> - 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.

---

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 async function GET(request: Request) {}
```

```
export async function GET(request) {}
```

Route Handlers can be nested anywhere 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 the native Request and Response APIs, Next.js extends them with `NextRequest` and `NextResponse` to provide convenient helpers for advanced use cases.

Behavior

Caching

Route Handlers are not cached by default. You can, however, opt into caching for `GET` methods. Other supported HTTP methods are **not** cached. To cache a `GET` method, use a route config option such as `export const dynamic = 'force-static'` in your Route Handler file.

```
export const dynamic = 'force-static'

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 const dynamic = 'force-static'

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 })
}
```

> **Good to know**: Other supported HTTP methods are **not** cached, even if they are placed alongside a `GET` method that is cached, in the same file.

Special Route Handlers

Special Route Handlers like `sitemap.ts`, `opengraph-image.tsx`, and `icon.tsx`, and other metadata files remain static by default unless they use Dynamic APIs or dynamic config options.

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 Incremental Static Regeneration (ISR):

```
export const revalidate = 60

export async function GET() {
  const data = await fetch('https://api.vercel.app/blog')
  const posts = await data.json()

  return Response.json(posts)
}
```

```
export const revalidate = 60

export async function GET() {
  const data = await fetch('https://api.vercel.app/blog')
  const posts = await data.json()

  return Response.json(posts)
}
```

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 = await 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 = await 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 = await 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 = await 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: Promise<{ slug: string }> }
) {
  const slug = (await params).slug // 'a', 'b', or 'c'
}
```

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

Route	Example URL	`params`
app/items/[slug]/route.js	/items/a	Promise<{ slug: 'a' }>
app/items/[slug]/route.js	/items/b	Promise<{ slug: 'b' }>
app/items/[slug]/route.js	/items/c	Promise<{ 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 '@ai-sdk/openai'
import { StreamingTextResponse, streamText } from 'ai'

export async function POST(req: Request) {
  const { messages } = await req.json()
  const result = await streamText({
    model: openai('gpt-4-turbo'),
    messages,
  })

  return new StreamingTextResponse(result.toAIStream())
}
```

```
import { openai } from '@ai-sdk/openai'
import { StreamingTextResponse, streamText } from 'ai'

export async function POST(req) {
  const { messages } = await req.json()
  const result = await streamText({
    model: openai('gpt-4-turbo'),
    messages,
  })

  return new StreamingTextResponse(result.toAIStream())
}
```

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 for a specific Route Handler using the standard Web API methods:

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

Good to know:

- To add CORS headers to multiple Route Handlers, you can use Middleware or the `next.config.js` file.
- Alternatively, see our CORS example package.

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.

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 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 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.

Use Cases

Integrating Middleware into your application can lead to significant improvements in performance, security, and user experience. Some common scenarios where Middleware is particularly effective include:

- Authentication and Authorization: Ensure user identity and check session cookies before granting access to specific pages or API routes.
- Server-Side Redirects: Redirect users at the server level based on certain conditions (e.g., locale, user role).
- Path Rewriting: Support A/B testing, feature rollouts, or legacy paths by dynamically rewriting paths to API routes or pages based on request properties.
- Bot Detection: Protect your resources by detecting and blocking bot traffic.
- Logging and Analytics: Capture and analyze request data for insights before processing by the page or API.
- Feature Flagging: Enable or disable features dynamically for seamless feature rollouts or testing.

Recognizing situations where middleware may not be the optimal approach is just as crucial. Here are some scenarios to be mindful of:

- Complex Data Fetching and Manipulation: Middleware is not designed for direct data fetching or manipulation, this should be done within Route Handlers or server-side utilities instead.
- Heavy Computational Tasks: Middleware should be lightweight and respond quickly or it can cause delays in page load. Heavy computational tasks or long-running processes should be done within dedicated Route Handlers.
- Extensive Session Management: While Middleware can manage basic session tasks, extensive session management should be managed by dedicated authentication services or within Route Handlers.
- Direct Database Operations: Performing direct database operations within Middleware is not recommended. Database interactions should be done within Route Handlers or server-side utilities.

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.

> **Note**: While only one `middleware.ts` file is supported per project, you can still organize your middleware logic modularly. Break out middleware functionalities into separate `.ts` or `.js` files and import them into your main `middleware.ts` file. This allows for cleaner management of route-specific middleware, aggregated in the `middleware.ts` for centralized control. By enforcing a single middleware file, it simplifies configuration, prevents potential conflicts, and optimizes performance by avoiding multiple middleware layers.

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**. Given this, it's crucial to use matchers to precisely target or exclude specific routes. 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, sitemap.xml, robots.txt (metadata files)
     */
    '/((?!api|_next/static|_next/image|favicon.ico|sitemap.xml|robots.txt).*)',
  ],
}
```

You can also bypass Middleware for certain requests by using the `missing` or `has` arrays, or a combination of both:

```
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, sitemap.xml, robots.txt (metadata files)
     */
    {
      source:
        '/((?!api|_next/static|_next/image|favicon.ico|sitemap.xml|robots.txt).*)',
      missing: [
        { type: 'header', key: 'next-router-prefetch' },
        { type: 'header', key: 'purpose', value: 'prefetch' },
      ],
    },

    {
      source:
        '/((?!api|_next/static|_next/image|favicon.ico|sitemap.xml|robots.txt).*)',
      has: [
        { type: 'header', key: 'next-router-prefetch' },
        { type: 'header', key: 'purpose', value: 'prefetch' },
      ],
    },

    {
      source:
        '/((?!api|_next/static|_next/image|favicon.ico|sitemap.xml|robots.txt).*)',
      has: [{ type: 'header', key: 'x-present' }],
      missing: [{ type: 'header', key: 'x-missing', 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.next
  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.next
  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.

CORS

You can set CORS headers in Middleware to allow cross-origin requests, including simple and preflighted requests.

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

const allowedOrigins = ['https://acme.com', 'https://my-app.org']

const corsOptions = {
  'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
  'Access-Control-Allow-Headers': 'Content-Type, Authorization',
}

export function middleware(request: NextRequest) {
  // Check the origin from the request
  const origin = request.headers.get('origin') ?? ''
  const isAllowedOrigin = allowedOrigins.includes(origin)

  // Handle preflighted requests
  const isPreflight = request.method === 'OPTIONS'

  if (isPreflight) {
    const preflightHeaders = {
      ...(isAllowedOrigin && { 'Access-Control-Allow-Origin': origin }),
      ...corsOptions,
    }
    return NextResponse.json({}, { headers: preflightHeaders })
  }

  // Handle simple requests
  const response = NextResponse.next()

  if (isAllowedOrigin) {
    response.headers.set('Access-Control-Allow-Origin', origin)
  }

  Object.entries(corsOptions).forEach(([key, value]) => {
    response.headers.set(key, value)
  })

  return response
}

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

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

const allowedOrigins = ['https://acme.com', 'https://my-app.org']

const corsOptions = {
  'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
  'Access-Control-Allow-Headers': 'Content-Type, Authorization',
}

export function middleware(request) {
  // Check the origin from the request
  const origin = request.headers.get('origin') ?? ''
  const isAllowedOrigin = allowedOrigins.includes(origin)

  // Handle preflighted requests
  const isPreflight = request.method === 'OPTIONS'

  if (isPreflight) {
    const preflightHeaders = {
      ...(isAllowedOrigin && { 'Access-Control-Allow-Origin': origin }),
      ...corsOptions,
    }
    return NextResponse.json({}, { headers: preflightHeaders })
  }
```

```
  // Handle simple requests
  const response = NextResponse.next()

  if (isAllowedOrigin) {
    response.headers.set('Access-Control-Allow-Origin', origin)
  }

  Object.entries(corsOptions).forEach(([key, value]) => {
    response.headers.set(key, value)
  })

  return response
}

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

Good to know: You can configure CORS headers for individual routes in Route Handlers.

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 type { 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+)/)
```

```
  ) {
    return NextResponse.redirect(
      new URL(`${req.nextUrl.pathname}/`, req.nextUrl)
    )
  }
}
```

`skipMiddlewareUrlNormalize` allows for disabling the URL normalization in Next.js to make handling direct visits and client-transitions the same. In some advanced cases, this option provides full control by using the original URL.

```
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
}
```

Unit Testing (experimental)

Starting in Next.js 15.1, the `next/experimental/testing/server` package contains utilities to help unit test middleware files. Unit testing middleware can help ensure that it's only run on desired paths and that custom routing logic works as intended before code reaches production.

The `unstable_doesMiddlewareMatch` function can be used to assert whether middleware will run for the provided URL, headers, and cookies.

```
import { unstable_doesMiddlewareMatch } from 'next/experimental/testing/server'

expect(
  unstable_doesMiddlewareMatch({
    config,
    nextConfig,
    url: '/test',
  })
).toEqual(false)
```

The entire middleware function can also be tested.

```
import { isRewrite, getRewrittenUrl } from 'next/experimental/testing/server'

const request = new NextRequest('https://nextjs.org/docs')
const response = await middleware(request)
expect(isRewrite(response)).toEqual(true)
expect(getRewrittenUrl(response)).toEqual('https://other-domain.com/docs')
// getRedirectUrl could also be used if the response were a redirect
```

Runtime

Middleware currently only supports APIs compatible with the Edge runtime. APIs exclusive to Node.js are unsupported.

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: 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.

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

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 NextResponse.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`
- `paraglide-next`
- `lingui`

---

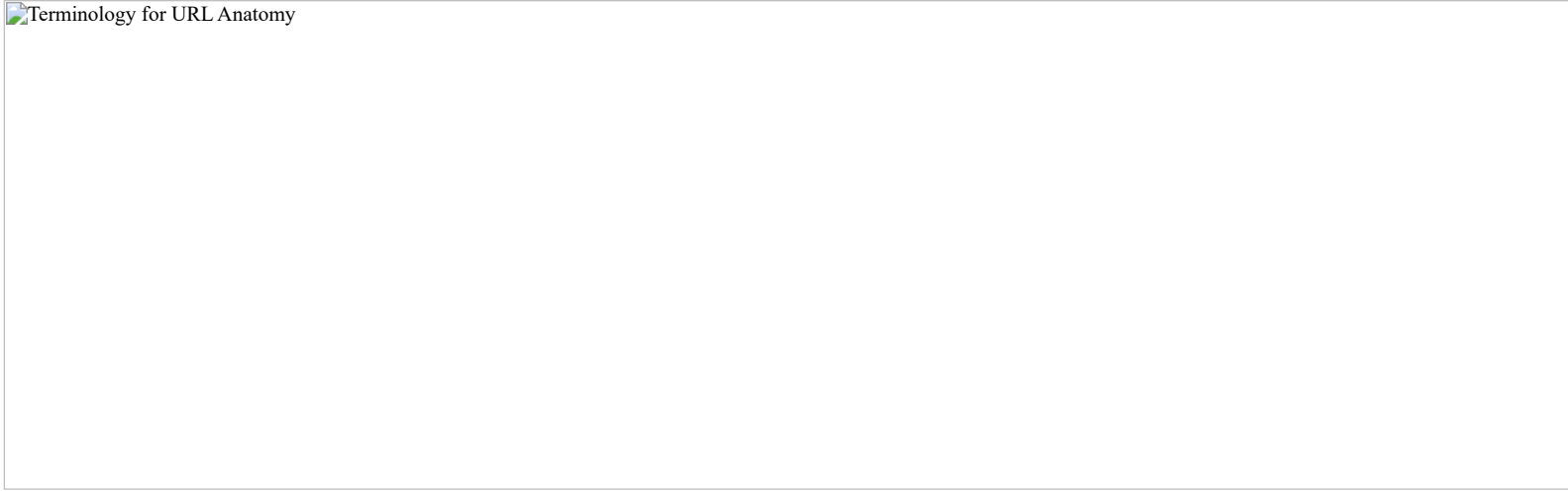
title: Routing Fundamentals nav_title: Routing description: Learn the fundamentals of routing for front-end applications.

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.



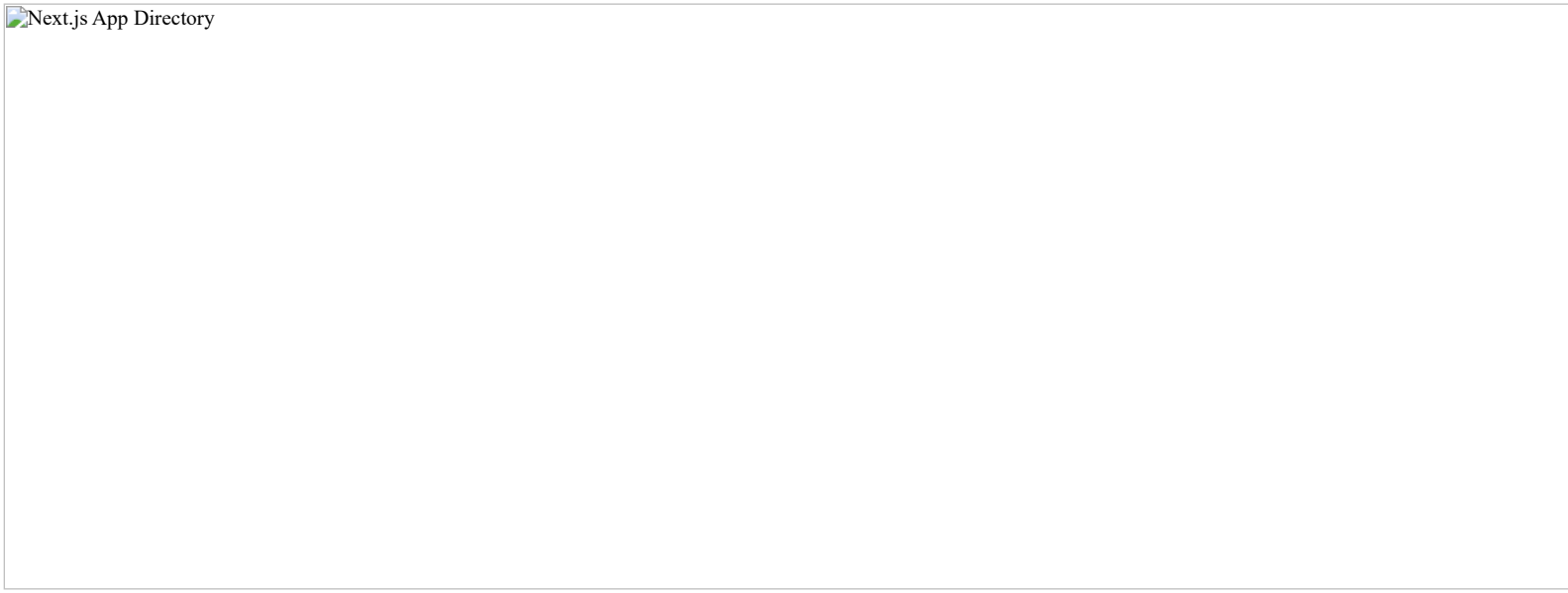
- **URL Segment:** Part of the URL path delimited by slashes.
- **URL Path:** Part of the URL that comes after the domain (composed of segments).

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:

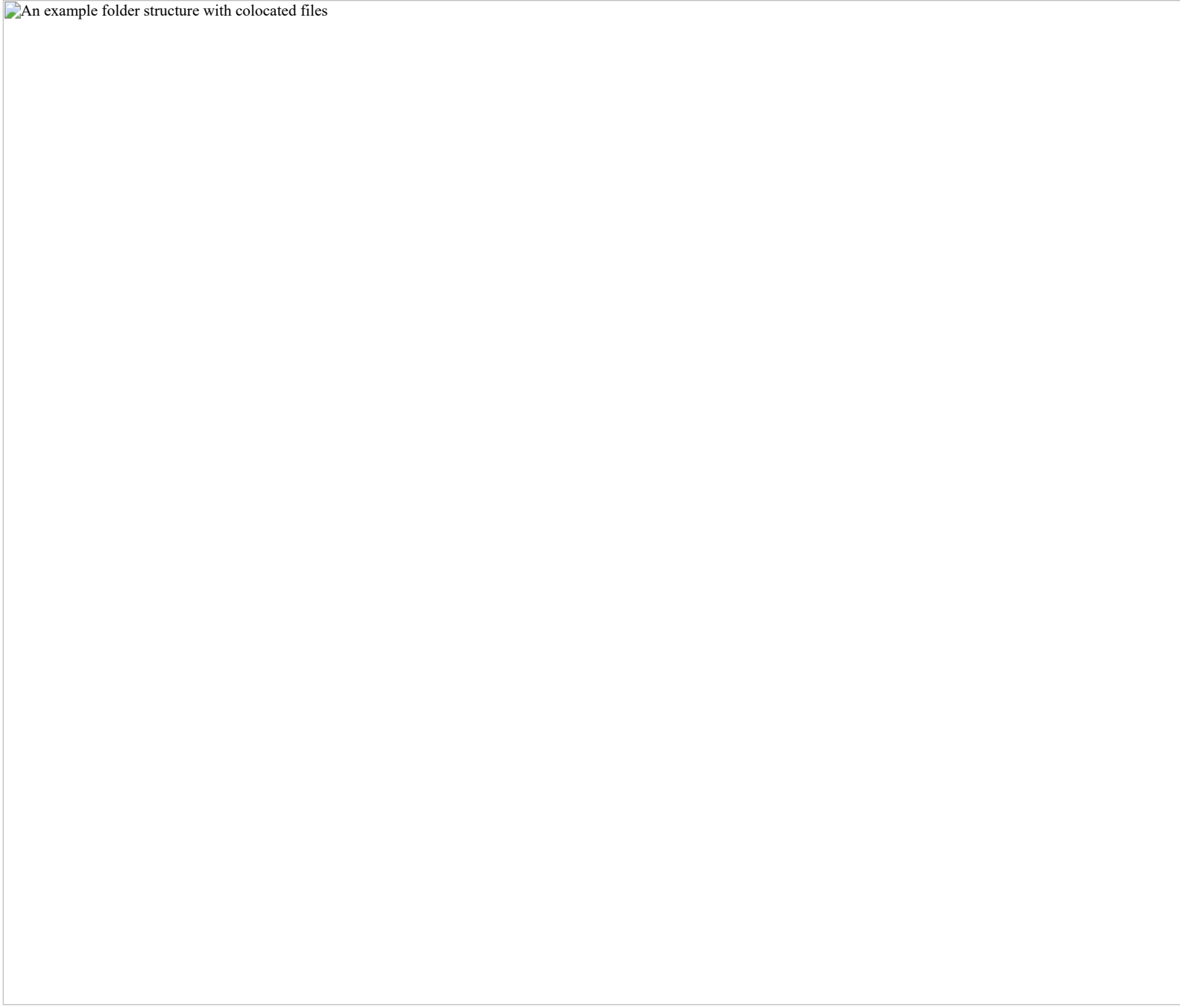
- `layout.js`
- `template.js`
- `error.js` (React error boundary)
- `loading.js` (React suspense boundary)
- `not-found.js` (React error boundary)
- `page.js` or nested `layout.js`

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 and Caching nav_title: Data Fetching and Caching description: Learn best practices for fetching data on the server or client in Next.js.

▶ Examples

This guide will walk you through the basics of data fetching and caching in Next.js, providing practical examples and best practices.

Here's a minimal example of data fetching in Next.js:

```
export default async function Page() {
  let data = await fetch('https://api.vercel.app/blog')
  let posts = await data.json()
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
```

```
  )
}

export default async function Page() {
  let data = await fetch('https://api.vercel.app/blog')
  let posts = await data.json()
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

This example demonstrates a basic server-side data fetch using the `fetch` API in an asynchronous React Server Component.

Reference

- `fetch`
- React `cache`
- Next.js `unstable_cache`

Examples

Fetching data on the server with the `fetch` API

This component will fetch and display a list of blog posts. The response from `fetch` is not cached by default.

```
export default async function Page() {
  let data = await fetch('https://api.vercel.app/blog')
  let posts = await data.json()
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

```
export default async function Page() {
  let data = await fetch('https://api.vercel.app/blog')
  let posts = await data.json()
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

If you are not using any Dynamic APIs anywhere else in this route, it will be prerendered during `next build` to a static page. The data can then be updated using Incremental Static Regeneration.

To prevent the page from prerendering, you can add the following to your file:

```
export const dynamic = 'force-dynamic'
```

However, you will commonly use functions like `cookies`, `headers`, or reading the incoming `searchParams` from the page props, which will automatically make the page render dynamically. In this case, you do *not* need to explicitly use `force-dynamic`.

Fetching data on the server with an ORM or database

This component will fetch and display a list of blog posts. The response from the database is not cached by default but could be with additional configuration.

```
import { db, posts } from '@/lib/db'

export default async function Page() {
  let allPosts = await db.select().from(posts)
  return (
    <ul>
      {allPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

```
import { db, posts } from '@/lib/db'

export default async function Page() {
  let allPosts = await db.select().from(posts)
  return (
    <ul>
      {allPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

If you are not using any Dynamic APIs anywhere else in this route, it will be prerendered during `next build` to a static page. The data can then be updated using Incremental Static Regeneration.

To prevent the page from prerendering, you can add the following to your file:

```
export const dynamic = 'force-dynamic'
```

However, you will commonly use functions like `cookies`, `headers`, or reading the incoming `searchParams` from the page props, which will automatically make the page render dynamically. In this case, you do *not* need to explicitly use `force-dynamic`.

Fetching data on the client

We recommend first attempting to fetch data on the server-side.

However, there are still cases where client-side data fetching makes sense. In these scenarios, you can manually call `fetch` in a `useEffect` (not recommended), or lean on popular React libraries in the community (such as SWR or React Query) for client fetching.

```
'use client'

import { useState, useEffect } from 'react'

export function Posts() {
  const [posts, setPosts] = useState(null)

  useEffect(() => {
    async function fetchPosts() {
      let res = await fetch('https://api.vercel.app/blog')
      let data = await res.json()
      setPosts(data)
    }
    fetchPosts()
  }, [])

  if (!posts) return <div>Loading...</div>

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

```
'use client'

import { useState, useEffect } from 'react'

export function Posts() {
  const [posts, setPosts] = useState(null)

  useEffect(() => {
    async function fetchPosts() {
      let res = await fetch('https://api.vercel.app/blog')
      let data = await res.json()
      setPosts(data)
    }
    fetchPosts()
  }, [])

  if (!posts) return <div>Loading...</div>

  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

Caching data with an ORM or Database

You can use the `unstable_cache` API to cache the response to allow pages to be prerendered when running `next build`.

```
import { unstable_cache } from 'next/cache'
import { db, posts } from '@/lib/db'

const getPosts = unstable_cache(
  async () => {
    return await db.select().from(posts)
  },
  ['posts'],
  { revalidate: 3600, tags: ['posts'] }
)

export default async function Page() {
  const allPosts = await getPosts()

  return (
    <ul>
      {allPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}
```

```
import { unstable_cache } from 'next/cache'
import { db, posts } from '@/lib/db'

const getPosts = unstable_cache(
  async () => {
    return await db.select().from(posts)
  },
  ['posts'],
  { revalidate: 3600, tags: ['posts'] }
)

export default async function Page() {
  const allPosts = await getPosts()

  return (
    <ul>
      {allPosts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
```

```
  )
}
```

This example caches the result of the database query for 1 hour (3600 seconds). It also adds the cache tag `posts` which can then be invalidated with Incremental Static Regeneration.

Reusing data across multiple functions

Next.js uses APIs like `generateMetadata` and `generateStaticParams` where you will need to use the same data fetched in the `page`.

If you are using `fetch`, requests can be memoized by adding `cache: 'force-cache'`. This means you can safely call the same URL with the same options, and only one request will be made.

> **Good to know:**
>
> - In previous versions of Next.js, using `fetch` would have a default `cache` value of `force-cache`. This changed in version 15, to a default of `cache: no-store`.

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

interface Post {
  id: string
  title: string
  content: string
}

async function getPost(id: string) {
  let res = await fetch(`https://api.vercel.app/blog/${id}`, {
    cache: 'force-cache',
  })
  let post: Post = await res.json()
  if (!post) notFound()
  return post
}

export async function generateStaticParams() {
  let posts = await fetch('https://api.vercel.app/blog', {
    cache: 'force-cache',
  }).then((res) => res.json())

  return posts.map((post: Post) => ({
    id: post.id,
  }))
}

export async function generateMetadata({ params }: { params: { id: string } }) {
  let post = await getPost(params.id)

  return {
    title: post.title,
  }
}

export default async function Page({ params }: { params: { id: string } }) {
  let post = await getPost(params.id)

  return (
    <article>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </article>
  )
}
```

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

async function getPost(id) {
  let res = await fetch(`https://api.vercel.app/blog/${id}`)
  let post = await res.json()
  if (!post) notFound()
  return post
}

export async function generateStaticParams() {
  let posts = await fetch('https://api.vercel.app/blog').then((res) =>
    res.json()
  )

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

export async function generateMetadata({ params }) {
  let post = await getPost(params.id)

  return {
    title: post.title,
  }
}

export default async function Page({ params }) {
  let post = await getPost(params.id)

  return (
    <article>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </article>
  )
}
```

If you are *not* using `fetch`, and instead using an ORM or database directly, you can wrap your data fetch with the React `cache` function. This will de-duplicate and only make one query.

```
import { cache } from 'react'
import { db, posts, eq } from '@/lib/db' // Example with Drizzle ORM
import { notFound } from 'next/navigation'

export const getPost = cache(async (id) => {
  const post = await db.query.posts.findFirst({
```

```
    where: eq(posts.id, parseInt(id)),
  })

  if (!post) notFound()
  return post
})
```

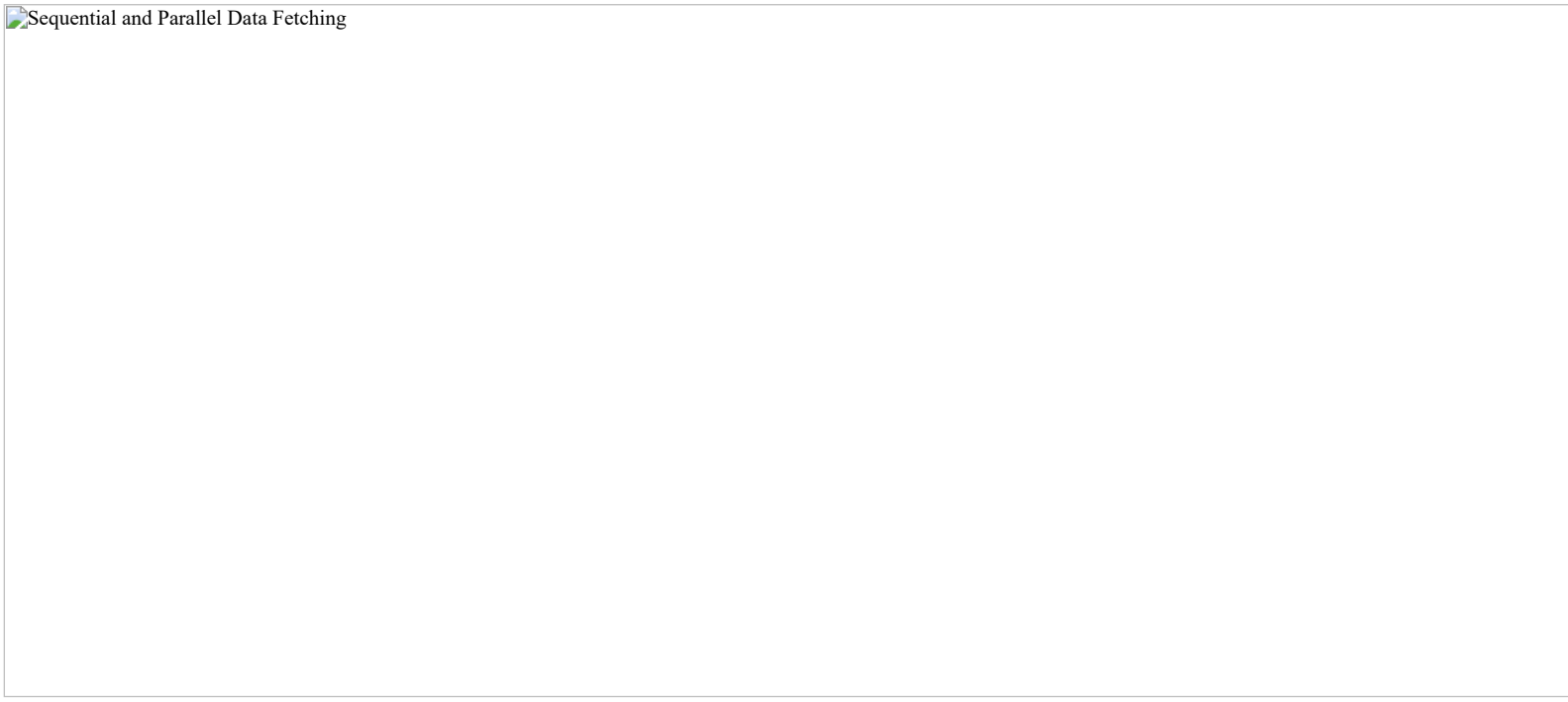
Revalidating cached data

Learn more about revalidating cached data with Incremental Static Regeneration.

Patterns

Parallel and sequential data fetching

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



- **Sequential**: requests in a component tree are dependent on each other. This can lead to longer loading times.
- **Parallel**: requests in a route are eagerly initiated and will load data at the same time. This reduces 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 not memoized.

There may be cases where you want this pattern because one fetch depends on the result of the other. 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:

```
export default async function Page({
  params: { username },
}: {
  params: { username: string }
}) {
  // Get artist information
  const artist = await getArtist(username)

  return (
    <>
      <h1>{artist.name}</h1>
      {/* Show fallback UI while the Playlists component is loading */}
      <Suspense fallback={<div>Loading...</div>}>
        {/* Pass the artist ID to the Playlists component */}
        <Playlists artistID={artist.id} />
      </Suspense>
    </>
  )
}

async function Playlists({ artistID }: { artistID: string }) {
  // Use the artist ID to fetch 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 } }) {
  // Get artist information
  const artist = await getArtist(username)

  return (
    <>
      <h1>{artist.name}</h1>
      {/* Show fallback UI while the Playlists component is loading */}
      <Suspense fallback={<div>Loading...</div>}>
        {/* Pass the artist ID to the Playlists component */}
```

```
        <Playlists artistID={artist.id} />
      </Suspense>
    </>
  )
}

async function Playlists({ artistID }) {
  // Use the artist ID to fetch playlists
  const playlists = await getArtistPlaylists(artistID)

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

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 requests, and the user will be able to interact with the parts of the page that are ready.

Parallel Data Fetching

By default, layout and page segments are rendered in parallel. This means requests will be initiated in parallel.

However, due to the nature of `async`/`await`, an awaited request inside the same segment or component will block any requests below it.

To fetch data in parallel, you can eagerly initiate requests by defining them outside the components that use the data. 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 `getAlbums` functions are defined outside the `Page` component and initiated inside the component using `Promise.all`:

```
import Albums from './albums'

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

async function getAlbums(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 }
}) {
  const artistData = getArtist(username)
  const albumsData = getAlbums(username)

  // Initiate both requests in parallel
  const [artist, albums] = await Promise.all([artistData, albumsData])

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

```
import Albums from './albums'

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

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

export default async function Page({ params: { username } }) {
  const artistData = getArtist(username)
  const albumsData = getAlbums(username)

  // Initiate both requests in parallel
  const [artist, albums] = await Promise.all([artistData, albumsData])

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

In addition, 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 by creating an utility function that you eagerly call above blocking requests. For example, `checkIsAvailable()` blocks `<Item/>` from rendering, so you can call `preload()` before it to eagerly initiate `<Item/>` data dependencies. By the time `<Item/>` is rendered, its data has already been fetched.

Note that `preload` function doesn't block `checkIsAvailable()` from running.

```
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
}
```

> **Good to know:** The "preload" function can also have any name as it's a pattern, not an API.

Using React `cache` and `server-only` with 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 address to the client",
```

```
    data,
    data.address
  )
  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 address to the client",
    data,
    data.address
  )
  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
      address={userData.address} // this will cause an error because of taintUniqueValue
    />
  )
}

import { getUserData } from './data'

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

---

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 called in Server and Client Components to handle form submissions and data mutations in Next.js applications.

> 📹 **Watch:** Learn more about 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:

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

  return '...'
}

export default function Page() {
  // Server Action
  async function create() {
    'use server'
    // Mutate data
  }

  return '...'
}
```

Client Components

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 exported 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() {}

'use client'
```

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

export function Button() {
  return <button onClick={() => create()}>Create</button>
}
```

```
'use client'

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

export function Button() {
  return <button onClick={() => create()}>Create</button>
}
```

Passing actions as props

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

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

```
'use client'

export default function ClientComponent({
  updateItemAction,
}: {
  updateItemAction: (formData: FormData) => void
}) {
  return <form action={updateItemAction}>{/* ... */}</form>
}
```

```
'use client'

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

Usually, the Next.js TypeScript plugin would flag `updateItemAction` in `client-component.tsx` since it is a function which generally can't be serialized across client-server boundaries. However, props named `action` or ending with `Action` are assumed to receive Server Actions. This is only a heuristic since the TypeScript plugin doesn't actually know if it receives a Server Action or an ordinary function. Runtime type-checking will still ensure you don't accidentally pass a function to a Client Component.

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.
- Server Actions inherit the Route Segment Config from the page or layout they are used on, including fields like `maxDuration`.

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)`. One thing to note is that the `formData` will include additional `$ACTION_` properties.

- 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: string, formData: FormData) {}
```

```
'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.

Nested form elements

You can also 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 programmatically using the `requestSubmit()` method. For example, when the user submits a form using the ⌘ + `Enter` keyboard shortcut, 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.

Server-side form validation

You can use the HTML attributes 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'

import { redirect } from 'next/navigation'

export async function createUser(prevState: any, formData: FormData) {
  const res = await fetch('https://...')
  const json = await res.json()

  if (!res.ok) {
    return { message: 'Please enter a valid email' }
  }

  redirect('/dashboard')
}
```

```
'use server'

import { redirect } from 'next/navigation'

export async function createUser(prevState, formData) {
  const res = await fetch('https://...')
  const json = await res.json()

  if (!res.ok) {
    return { message: 'Please enter a valid email' }
  }

  redirect('/dashboard')
}
```

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">{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">{state?.message}</p>
      <button>Sign up</button>
    </form>
  )
}
```

Good to know:

- These examples use React's `useFormState` hook, which is bundled with the Next.js App Router. If you are using React 19, use `useActionState` instead. See the React docs for more information.

Pending states

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

The `useFormStatus` hook exposes a `pending` boolean that can be used to show a loading indicator while the action is being executed.

```
'use client'

import { useFormStatus } from 'react-dom'

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

  return (
    <button disabled={pending} type="submit">
      Sign Up
    </button>
  )
}
```

```
'use client'

import { useFormStatus } from 'react-dom'

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

  return (
    <button disabled={pending} type="submit">
      Sign Up
    </button>
  )
}
```

Good to know:

- In React 19, `useFormStatus` includes additional keys on the returned object, like data, method, and action. If you are not using React 19, only the `pending` key is available.
- In React 19, `useActionState` also includes a `pending` key on the returned state.

Optimistic updates

You can use the React `useOptimistic` hook to optimistically update the UI before the Server Action finishes executing, 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[],
    string
  >(messages, (state, newMessage) => [...state, { message: newMessage }])

  const formAction = async (formData) => {
    const message = formData.get('message') as string
    addOptimisticMessage(message)
    await send(message)
  }

  return (
    <div>
      {optimisticMessages.map((m, i) => (
        <div key={i}>{m.message}</div>
      ))}
      <form action={formAction}>
        <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 }]
  )

  const formAction = async (formData) => {
    const message = formData.get('message')
    addOptimisticMessage(message)
    await send(message)
  }

  return (
    <div>
      {optimisticMessages.map((m) => (
        <div>{m.message}</div>
      ))}
      <form action={formAction}>
        <input type="text" name="message" />
        <button type="submit">Send</button>
      </form>
    </div>
  )
}
```

Event handlers

While it's common to use Server Actions within `<form>` elements, they can also be invoked with event handlers such as `onClick`. For example, to increment a like count:

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

```
'use client'

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

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

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

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>
}
```

```
'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() {
  const cookieStore = await cookies()

  // Get cookie
  cookieStore.get('name')?.value

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

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

```
'use server'

import { cookies } from 'next/headers'

export async function exampleAction() {
  // Get cookie
  const cookieStore = await cookies()

  // Get cookie
  cookieStore.get('name')?.value

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

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

See additional examples for deleting cookies from Server Actions.

Security

By default, when a Server Action is created and exported, it creates a public HTTP endpoint and should be treated with the same security assumptions and authorization checks. This means, even if a Server Action or utility function is not imported elsewhere in your code, it's still a publicly accessible.

To improve security, Next.js has the following built-in features:

- **Secure action IDs:** Next.js creates encrypted, non-deterministic IDs to allow the client to reference and call the Server Action. These IDs are periodically recalculated between builds for enhanced security.
- **Dead code elimination:** Unused Server Actions (referenced by their IDs) are removed from client bundle to avoid public access by third-party.

  Good to know:

  The IDs are created during compilation and are cached for a maximum of 14 days. They will be regenerated when a new build is initiated or when the build cache is invalidated. This security improvement reduces the risk in cases where an authentication layer is missing. However, you should still treat Server Actions like public HTTP endpoints.

```
// app/actions.js
'use server'

// This action **is** used in our application, so Next.js
// will create a secure ID to allow the client to reference
// and call the Server Action.
export async function updateUserAction(formData) {}

// This action **is not** used in our application, so Next.js
// will automatically remove this code during `next build`
// and will not create a public endpoint.
export async function deleteUserAction(formData) {}
```

Authentication and authorization

You should 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 async 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 (
    <form>
      <button formAction={publish}>Publish</button>
    </form>
  );
}
```

```
export default async 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 (
    <form>
      <button formAction={publish}>Publish</button>
    </form>
  );
}
```

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, check out the following React docs:

- Server Actions
- `"use server"`
- `<form>`
- `useFormStatus`
- `useActionState`
- `useOptimistic`

---

title: Incremental Static Regeneration (ISR) description: Learn how to create or update static pages at runtime with Incremental Static Regeneration.

▶ Examples

Incremental Static Regeneration (ISR) enables you to:

- Update static content without rebuilding the entire site
- Reduce server load by serving prerendered, static pages for most requests
- Ensure proper `cache-control` headers are automatically added to pages
- Handle large amounts of content pages without long `next build` times

Here's a minimal example:

```
interface Post {
  id: string
  title: string
  content: string
}

// Next.js will invalidate the cache when a
// request comes in, at most once every 60 seconds.
export const revalidate = 60

// We'll prerender only the params from `generateStaticParams` at build time.
// If a request comes in for a path that hasn't been generated,
// Next.js will server-render the page on-demand.
export const dynamicParams = true // or false, to 404 on unknown paths

export async function generateStaticParams() {
  const posts: Post[] = await fetch('https://api.vercel.app/blog').then((res) =>
    res.json()
  )
  return posts.map((post) => ({
    id: String(post.id),
  }))
}

export default async function Page({ params }: { params: { id: string } }) {
  const post: Post = await fetch(
    `https://api.vercel.app/blog/${params.id}`
  ).then((res) => res.json())
  return (
    <main>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </main>
  )
}
```

```
// Next.js will invalidate the cache when a
// request comes in, at most once every 60 seconds.
export const revalidate = 60

// We'll prerender only the params from `generateStaticParams` at build time.
// If a request comes in for a path that hasn't been generated,
// Next.js will server-render the page on-demand.
export const dynamicParams = true // or false, to 404 on unknown paths

export async function generateStaticParams() {
  const posts = await fetch('https://api.vercel.app/blog').then((res) =>
    res.json()
  )
  return posts.map((post) => ({
```

```
    id: String(post.id),
  }))
}

export default async function Page({ params }) {
  const post = await fetch(`https://api.vercel.app/blog/${params.id}`).then(
    (res) => res.json()
  )
  return (
    <main>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </main>
  )
}
```

```
import type { GetStaticPaths, GetStaticProps } from 'next'

interface Post {
  id: string
  title: string
  content: string
}

interface Props {
  post: Post
}

export const getStaticPaths: GetStaticPaths = async () => {
  const posts = await fetch('https://api.vercel.app/blog').then((res) =>
    res.json()
  )
  const paths = posts.map((post: Post) => ({
    params: { id: String(post.id) },
  }))

  // We'll prerender only these paths at build time.
  // { fallback: 'blocking' } will server-render pages
  // on-demand if the path doesn't exist.
  return { paths, fallback: false }
}

export const getStaticProps: GetStaticProps<Props> = async ({
  params,
}: {
  params: { id: string }
}) => {
  const post = await fetch(`https://api.vercel.app/blog/${params.id}`).then(
    (res) => res.json()
  )

  return {
    props: { post },
    // Next.js will invalidate the cache when a
    // request comes in, at most once every 60 seconds.
    revalidate: 60,
  }
}

export default function Page({ post }: Props) {
  return (
    <main>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </main>
  )
}
```

```
export async function getStaticPaths() {
  const posts = await fetch('https://api.vercel.app/blog').then((res) =>
    res.json()
  )
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // We'll prerender only these paths at build time.
  // { fallback: false } means other routes should 404.
  return { paths, fallback: false }
}

export async function getStaticProps({ params }) {
  const post = await fetch(`https://api.vercel.app/blog/${params.id}`).then(
    (res) => res.json()
  )

  return {
    props: { post },
    // Next.js will invalidate the cache when a
    // request comes in, at most once every 60 seconds.
    revalidate: 60,
  }
}

export default function Page({ post }) {
  return (
    <main>
      <h1>{post.title}</h1>
      <p>{post.content}</p>
    </main>
  )
}
```

Here's how this example works:

1. During `next build`, all known blog posts are generated (there are 25 in this example)
2. All requests made to these pages (e.g. `/blog/1`) are cached and instantaneous
3. After 60 seconds has passed, the next request will still show the cached (stale) page
4. The cache is invalidated and a new version of the page begins generating in the background
5. Once generated successfully, Next.js will display and cache the updated page
6. If `/blog/26` is requested, Next.js will generate and cache this page on-demand

Reference

Route segment config

- revalidate
- dynamicParams

Functions

- revalidatePath
- revalidateTag

Functions

- getStaticProps
- res.revalidate

Examples

Time-based revalidation

This fetches and displays a list of blog posts on `/blog`. After an hour, the cache for this page is invalidated on the next visit to the page. Then, in the background, a new version of the page is generated with the latest blog posts.

```
interface Post {
  id: string
  title: string
  content: string
}

export const revalidate = 3600 // invalidate every hour

export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog')
  const posts: Post[] = await data.json()
  return (
    <main>
      <h1>Blog Posts</h1>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </main>
  )
}
```

```
export const revalidate = 3600 // invalidate every hour

export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog')
  const posts = await data.json()
  return (
    <main>
      <h1>Blog Posts</h1>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </main>
  )
}
```

We recommend setting a high revalidation time. For instance, 1 hour instead of 1 second. If you need more precision, consider using on-demand revalidation. If you need real-time data, consider switching to dynamic rendering.

On-demand revalidation with `revalidatePath`

For a more precise method of revalidation, invalidate pages on-demand with the `revalidatePath` function.

For example, this Server Action would get called after adding a new post. Regardless of how you retrieve your data in your Server Component, either using `fetch` or connecting to a database, this will clear the cache for the entire route and allow the Server Component to fetch fresh data.

```
'use server'

import { revalidatePath } from 'next/cache'

export async function createPost() {
  // Invalidate the /posts route in the cache
  revalidatePath('/posts')
}
```

```
'use server'

import { revalidatePath } from 'next/cache'

export async function createPost() {
  // Invalidate the /posts route in the cache
  revalidatePath('/posts')
}
```

View a demo and explore the source code.

On-demand revalidation with `revalidateTag`

For most use cases, prefer revalidating entire paths. If you need more granular control, you can use the `revalidateTag` function. For example, you can tag individual `fetch` calls:

```
export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog', {
    next: { tags: ['posts'] },
```

```
  })
  const posts = await data.json()
  // ...
}

export default async function Page() {
  const data = await fetch('https://api.vercel.app/blog', {
    next: { tags: ['posts'] },
  })
  const posts = await data.json()
  // ...
}
```

If you are using an ORM or connecting to a database, you can use `unstable_cache`:

```
import { unstable_cache } from 'next/cache'
import { db, posts } from '@/lib/db'

const getCachedPosts = unstable_cache(
  async () => {
    return await db.select().from(posts)
  },
  ['posts'],
  { revalidate: 3600, tags: ['posts'] }
)

export default async function Page() {
  const posts = getCachedPosts()
  // ...
}
```


```
import { unstable_cache } from 'next/cache'
import { db, posts } from '@/lib/db'

const getCachedPosts = unstable_cache(
  async () => {
    return await db.select().from(posts)
  },
  ['posts'],
  { revalidate: 3600, tags: ['posts'] }
)

export default async function Page() {
  const posts = getCachedPosts()
  // ...
}
```


You can then use `revalidateTag` in a Server Actions or Route Handler:

```
'use server'

import { revalidateTag } from 'next/cache'

export async function createPost() {
  // Invalidate all data tagged with 'posts' in the cache
  revalidateTag('posts')
}
```


```
'use server'

import { revalidateTag } from 'next/cache'

export async function createPost() {
  // Invalidate all data tagged with 'posts' in the cache
  revalidateTag('posts')
}
```


On-demand validation with `res.revalidate()`

For a more precise method of revalidation, use `res.revalidate` to generate a new page on-demand from an API Router.

For example, this API Route can be called at `/api/revalidate?secret=<token>` to revalidate a given blog post. Create a secret token only known by your Next.js app. This secret will be used to prevent unauthorized access to the revalidation API Route.

```
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  // Check for secret to confirm this is a valid request
  if (req.query.secret !== process.env.MY_SECRET_TOKEN) {
    return res.status(401).json({ message: 'Invalid token' })
  }

  try {
    // This should be the actual path not a rewritten path
    // e.g. for "/posts/[id]" this should be "/posts/1"
    await res.revalidate('/posts/1')
    return res.json({ revalidated: true })
  } catch (err) {
    // If there was an error, Next.js will continue
    // to show the last successfully generated page
    return res.status(500).send('Error revalidating')
  }
}

export default async function handler(req, res) {
  // Check for secret to confirm this is a valid request
  if (req.query.secret !== process.env.MY_SECRET_TOKEN) {
    return res.status(401).json({ message: 'Invalid token' })
  }

  try {
    // This should be the actual path not a rewritten path
    // e.g. for "/posts/[id]" this should be "/posts/1"
    await res.revalidate('/posts/1')
    return res.json({ revalidated: true })
  } catch (err) {
    // If there was an error, Next.js will continue
```

```
    // to show the last successfully generated page
    return res.status(500).send('Error revalidating')
  }
}
```

If you are using on-demand revalidation, you do not need to specify a `revalidate` time inside of `getStaticProps`. Next.js will use the default value of `false` (no revalidation) and only revalidate the page on-demand when `res.revalidate()` is called.

Handling uncaught exceptions

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. Learn more about error handling.

If there is an error inside `getStaticProps` when handling background regeneration, or you manually throw an error, the last successfully generated page will continue to show. On the next subsequent request, Next.js will retry calling `getStaticProps`.

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

interface Post {
  id: string
  title: string
  content: string
}

interface Props {
  post: Post
}

export const getStaticProps: GetStaticProps<Props> = async ({
  params,
}: {
  params: { id: string }
}) => {
  // If this request throws an uncaught error, Next.js will
  // not invalidate the currently shown page and
  // retry getStaticProps on the next request.
  const res = await fetch(`https://api.vercel.app/blog/${params.id}`)
  const post: Post = await res.json()

  if (!res.ok) {
    // If there is a server error, you might want to
    // throw an error instead of returning so that the cache is not updated
    // until the next successful request.
    throw new Error(`Failed to fetch posts, received status ${res.status}`)
  }

  return {
    props: { post },
    // Next.js will invalidate the cache when a
    // request comes in, at most once every 60 seconds.
    revalidate: 60,
  }
}

export async function getStaticProps({ params }) {
  // If this request throws an uncaught error, Next.js will
  // not invalidate the currently shown page and
  // retry getStaticProps on the next request.
  const res = await fetch(`https://api.vercel.app/blog/${params.id}`)
  const post = await res.json()

  if (!res.ok) {
    // If there is a server error, you might want to
    // throw an error instead of returning so that the cache is not updated
    // until the next successful request.
    throw new Error(`Failed to fetch posts, received status ${res.status}`)
  }

  return {
    props: { post },
    // Next.js will invalidate the cache when a
    // request comes in, at most once every 60 seconds.
    revalidate: 60,
  }
}
```

Customizing the cache location

Caching and revalidating pages (with Incremental Static Regeneration) use the same shared cache. When deploying to Vercel, the ISR cache is automatically persisted to durable storage.

When self-hosting, the ISR 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. Learn more.

Troubleshooting

Debugging cached data in local development

If you are using the `fetch` API, you can add additional logging to understand which requests are cached or uncached. Learn more about the `logging` option.

```
module.exports = {
  logging: {
    fetches: {
      fullUrl: true,
    },
  },
}
```

Verifying correct production behavior

To verify your pages are cached and revalidated correctly in production, you can test locally by running `next build` and then `next start` to run the production Next.js server.

This will allow you to test ISR behavior as it would work in a production environment. For further debugging, add the following environment variable to your `.env` file:

```
NEXT_PRIVATE_DEBUG_CACHE=1
```

This will make the Next.js server console log ISR cache hits and misses. You can inspect the output to see which pages are generated during `next build`, as well as how pages are updated as paths are accessed on-demand.

Caveats

- ISR is only supported when using the Node.js runtime (default).
- ISR is not supported when creating a Static Export.
- If you have multiple `fetch` requests in a statically rendered route, and each has a different `revalidate` frequency, the lowest time will be used for ISR. However, those revalidate frequencies will still be respected by the Data Cache.
- If any of the `fetch` requests used on a route have a `revalidate` time of `0`, or an explicit `no-store`, the route will be dynamically rendered.
- Middleware won't be executed for on-demand ISR requests, meaning any path rewrites or logic in Middleware will not be applied. Ensure you are revalidating the exact path. For example, `/post/1` instead of a rewritten `/post-1`.

- ISR is only supported when using the Node.js runtime (default).
- ISR is not supported when creating a Static Export.
- Middleware won't be executed for on-demand ISR requests, meaning any path rewrites or logic in Middleware will not be applied. Ensure you are revalidating the exact path. For example, `/post/1` instead of a rewritten `/post-1`.

Version history

Version	Changes
`v14.1.0`	Custom `cacheHandler` is stable.
`v13.0.0`	App Router is introduced.
`v12.2.0`	Pages Router: On-Demand ISR is stable
`v12.0.0`	Pages Router: Bot-aware ISR fallback added.
`v9.5.0`	Pages Router: Stable ISR introduced.

---

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.
- **Performance**: Server Components give you additional tools to optimize performance from the baseline. For example, if you start with an app composed of entirely Client Components, moving non-interactive pieces of your UI to Server Components can reduce the amount of client-side JavaScript needed. This is beneficial for users with slower internet or less powerful devices, as the browser has less client-side JavaScript to download, parse, and execute.
- **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)

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

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 API or uncached data request is discovered, Next.js will switch to dynamically rendering the whole route. This table summarizes how Dynamic APIs and data caching affect whether a route is statically or dynamically rendered:

Dynamic APIs	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.

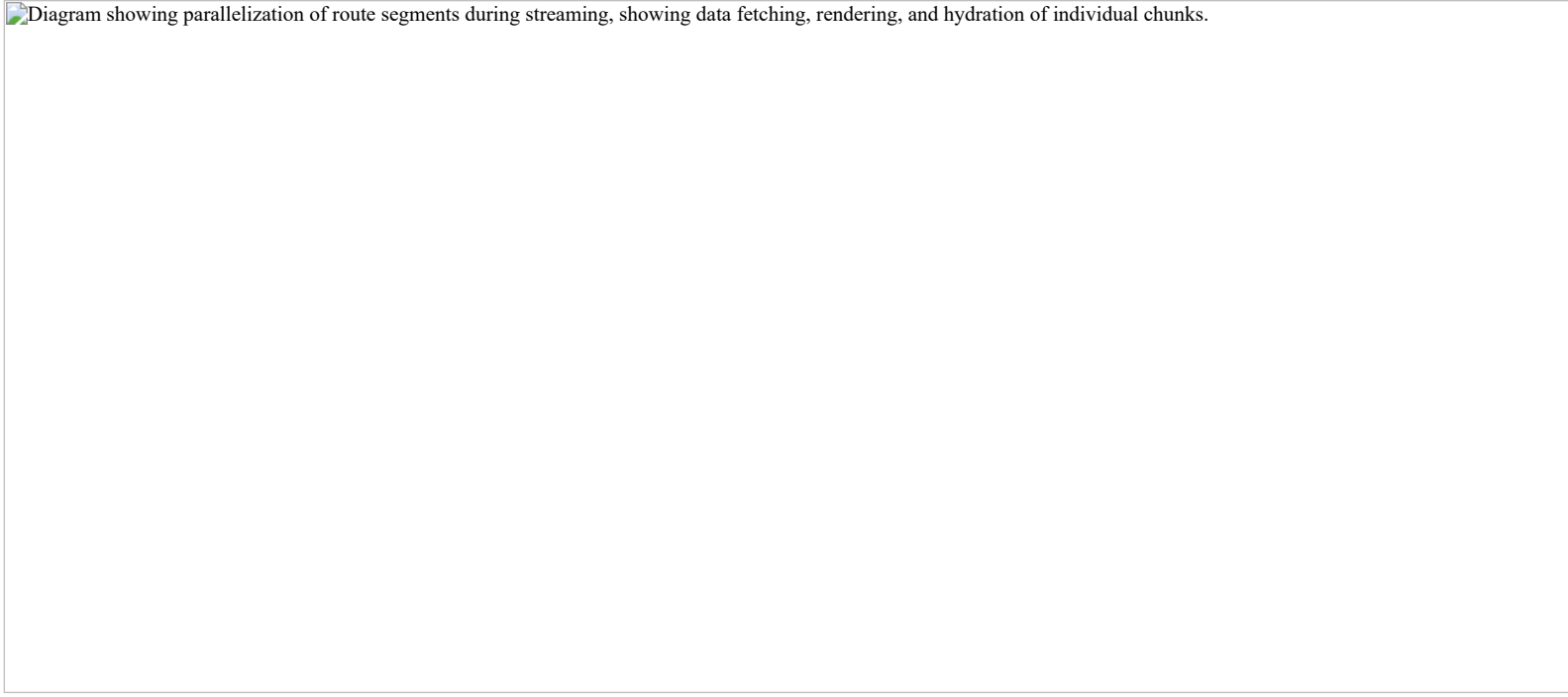
As a developer, you do not need to choose between static and dynamic rendering as Next.js will automatically choose the best rendering strategy for each route based on the features and APIs used. Instead, you choose when to cache or revalidate specific data, and you may choose to stream parts of your UI.

Dynamic APIs

Dynamic APIs rely on information that can only be known at request time (and not ahead of time during prerendering). Using any of these APIs signals the developer's intention and will opt the whole route into dynamic rendering at the request time. These APIs include:

- `cookies`
- `headers`
- `connection`
- `draftMode`
- `searchParams` prop
- `unstable_noStore`
- `unstable_after`

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 allow you to write interactive UI that is prerendered on the server and can use client JavaScript to run in the browser.

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.

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, all components in the App Router are Server Components where these APIs are not available. By defining the `"use client"` directive in `toggle.js`, you can tell React to enter the client boundary where these 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.

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.

View an example of this pattern.

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() {
  const [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() {
  const [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 with 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: Partial Prerendering description: Learn how to combine the benefits of static and dynamic rendering with Partial Prerendering.

> **Note:** Partial Prerendering is an **experimental** feature only available on canary and is subject to change. It is not ready for production use.

Partial Prerendering (PPR) enables you to combine static and dynamic components together in the same route.

During the build, Next.js prerenders as much of the route as possible. If dynamic code is detected, like reading from the incoming request, you can wrap the relevant component with a React Suspense boundary. The Suspense boundary fallback will then be included in the prerendered HTML.

> **Watch:** Why PPR and how it works → YouTube (10 minutes).

Background

PPR enables your Next.js server to immediately send prerendered content.

To prevent client to server waterfalls, dynamic components begin streaming from the server in parallel while serving the initial prerender. This ensures dynamic components can begin rendering before client JavaScript has been loaded in the browser.

To prevent creating many HTTP requests for each dynamic component, PPR is able to combine the static prerender and dynamic components together into a single HTTP request. This ensures there are not multiple network roundtrips needed for each dynamic component.

Using Partial Prerendering

Incremental Adoption (Version 15)

In Next.js 15, you can incrementally adopt Partial Prerendering in layouts and pages by setting the `ppr` option in `next.config.js` to `incremental`, and exporting the `experimental_ppr` route config option at the top of the file:

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

const nextConfig: NextConfig = {
  experimental: {
    ppr: 'incremental',
  },
}

export default nextConfig
```

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    ppr: 'incremental',
  },
}

module.exports = nextConfig
```

```
import { Suspense } from "react"
import { StaticComponent, DynamicComponent, Fallback } from "@/app/ui"

export const experimental_ppr = true

export default function Page() {
  return {
     <>
      <StaticComponent />
      <Suspense fallback={<Fallback />}>
        <DynamicComponent />
      </Suspense>
     </>
  };
}
```

```
import { Suspense } from "react"
import { StaticComponent, DynamicComponent, Fallback } from "@/app/ui"

export const experimental_ppr = true

export default function Page() {
  return {
     <>
      <StaticComponent />
      <Suspense fallback={<Fallback />}>
        <DynamicComponent />
      </Suspense>
     </>
  };
}
```

Good to know:

- Routes that don't have `experimental_ppr` will default to `false` and will not be prerendered using PPR. You need to explicitly opt-in to PPR for each route.
- `experimental_ppr` will apply to all children of the route segment, including nested layouts and pages. You don't have to add it to every file, only the top segment of a route.
- To disable PPR for children segments, you can set `experimental_ppr` to `false` in the child segment.

Dynamic Components

When creating the prerender for your route during `next build`, Next.js requires that Dynamic APIs are wrapped with React Suspense. The `fallback` is then included in the prerender.

For example, using functions like `cookies` or `headers`:

```
import { cookies } from 'next/headers'

export async function User() {
  const session = (await cookies()).get('session')?.value
  return '...'
}
```

```
import { cookies } from 'next/headers'

export async function User() {
  const session = (await cookies()).get('session')?.value
  return '...'
}
```

This component requires looking at the incoming request to read cookies. To use this with PPR, you should wrap the component with Suspense:

```
import { Suspense } from 'react'
import { User, AvatarSkeleton } from './user'

export const experimental_ppr = true

export default function Page() {
  return (
```

```
    <section>
      <h1>This will be prerendered</h1>
      <Suspense fallback={<AvatarSkeleton />}>
        <User />
      </Suspense>
    </section>
  )
}
```

```
import { Suspense } from 'react'
import { User, AvatarSkeleton } from './user'

export const experimental_ppr = true

export default function Page() {
  return (
    <section>
      <h1>This will be prerendered</h1>
      <Suspense fallback={<AvatarSkeleton />}>
        <User />
      </Suspense>
    </section>
  )
}
```

Components only opt into dynamic rendering when the value is accessed.

For example, if you are reading `searchParams` from a `page`, you can forward this value to another component as a prop:

```
import { Table } from './table'

export default function Page({
  searchParams,
}: {
  searchParams: Promise<{ sort: string }>
}) {
  return (
    <section>
      <h1>This will be prerendered</h1>
      <Table searchParams={searchParams} />
    </section>
  )
}
```

```
import { Table } from './table'

export default function Page({ searchParams }) {
  return (
    <section>
      <h1>This will be prerendered</h1>
      <Table searchParams={searchParams} />
    </section>
  )
}
```

Inside of the table component, accessing the value from `searchParams` will make the component run dynamically:

```
export async function Table({
  searchParams,
}: {
  searchParams: Promise<{ sort: string }>
}) {
  const sort = (await searchParams).sort === 'true'
  return '...'
}
```

```
export async function Table({ searchParams }) {
  const sort = (await searchParams).sort === 'true'
  return '...'
}
```

---

title: Runtimes description: Learn about the switchable runtimes (Edge and Node.js) in Next.js. related: description: View the Edge Runtime API reference. links: - app/api-reference/edge

{/* 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 two server runtimes you can use in your application:

- The **Node.js Runtime** (default), which has access to all Node.js APIs and compatible packages from the ecosystem.
- The **Edge Runtime** which contains a more limited set of APIs.

Use Cases

- The Node.js Runtime is used for rendering your application.
- The Edge Runtime is used for Middleware (routing rules like redirects, rewrites, and setting headers).

Caveats

- The Edge Runtime does not support all Node.js APIs. Some packages may not work as expected. Learn more about the unsupported APIs in the Edge Runtime.
- The Edge Runtime does not support Incremental Static Regeneration (ISR).
- Both runtimes can 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.

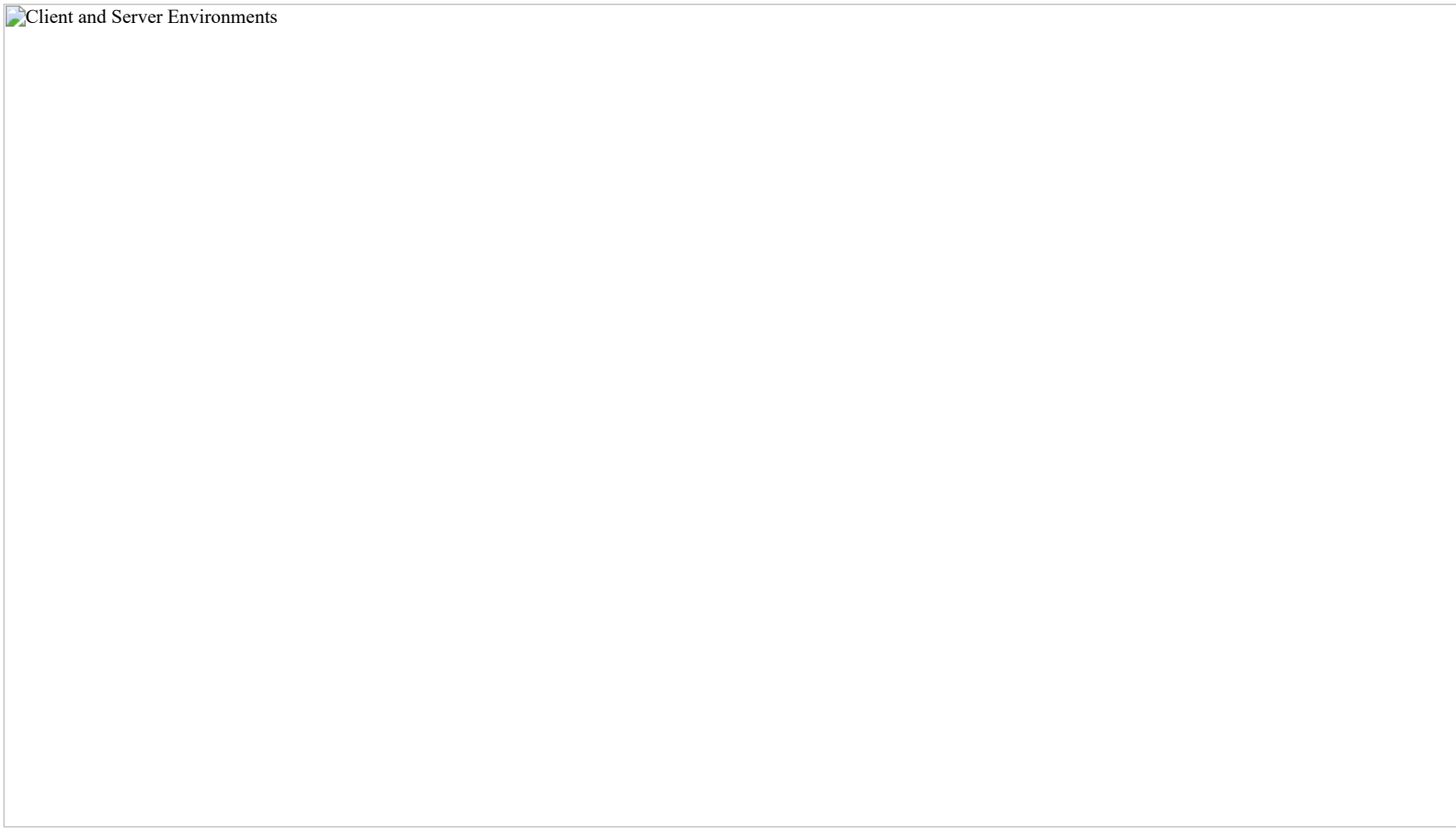
Fundamentals

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.

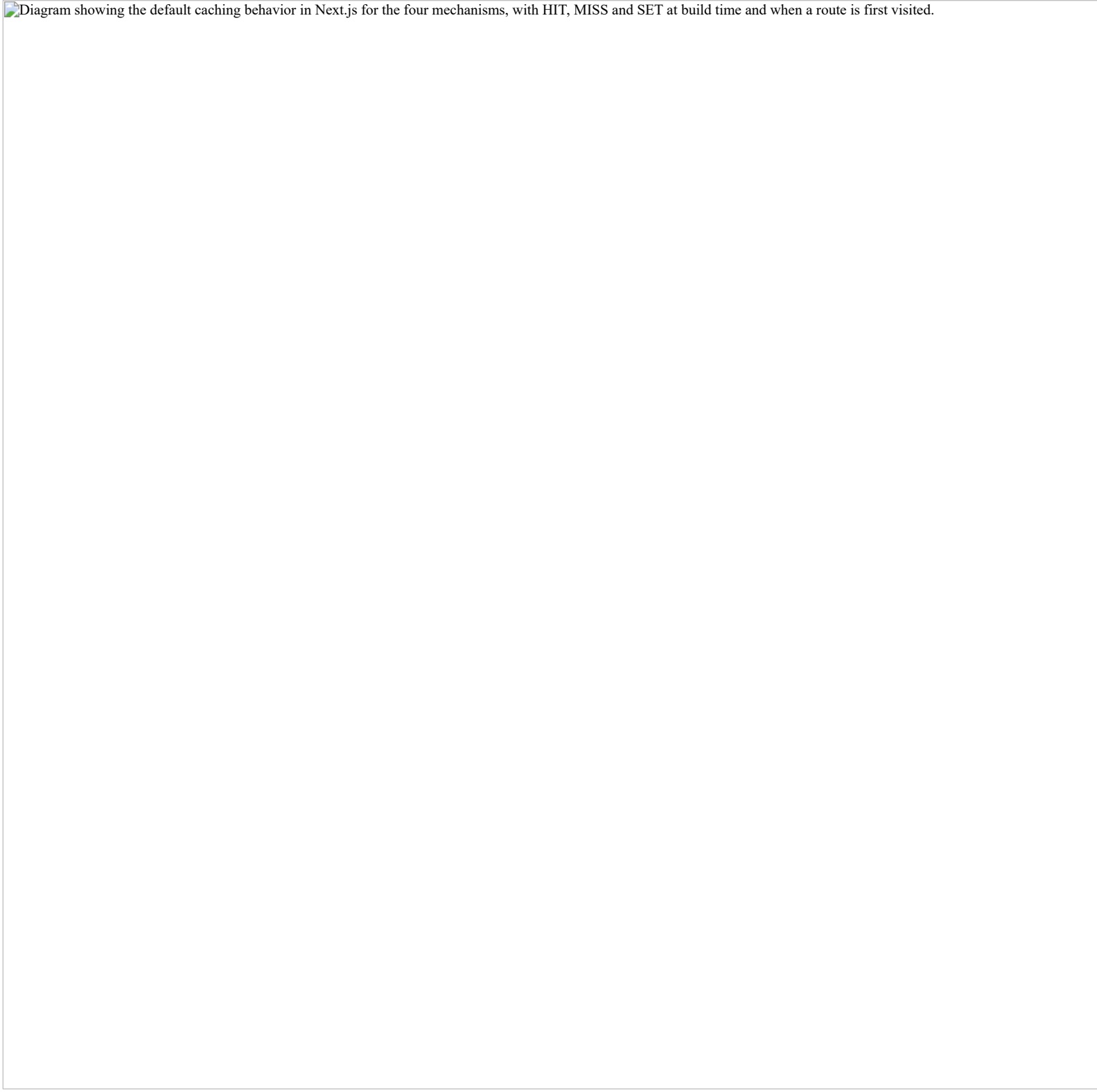
> **Good to know**: This page helps you understand how Next.js works under the hood but is **not** essential knowledge to be productive with Next.js. Most of Next.js' caching heuristics are determined by your API usage and have defaults for the best performance with zero or minimal configuration. If you instead want to jump to examples, start here.

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

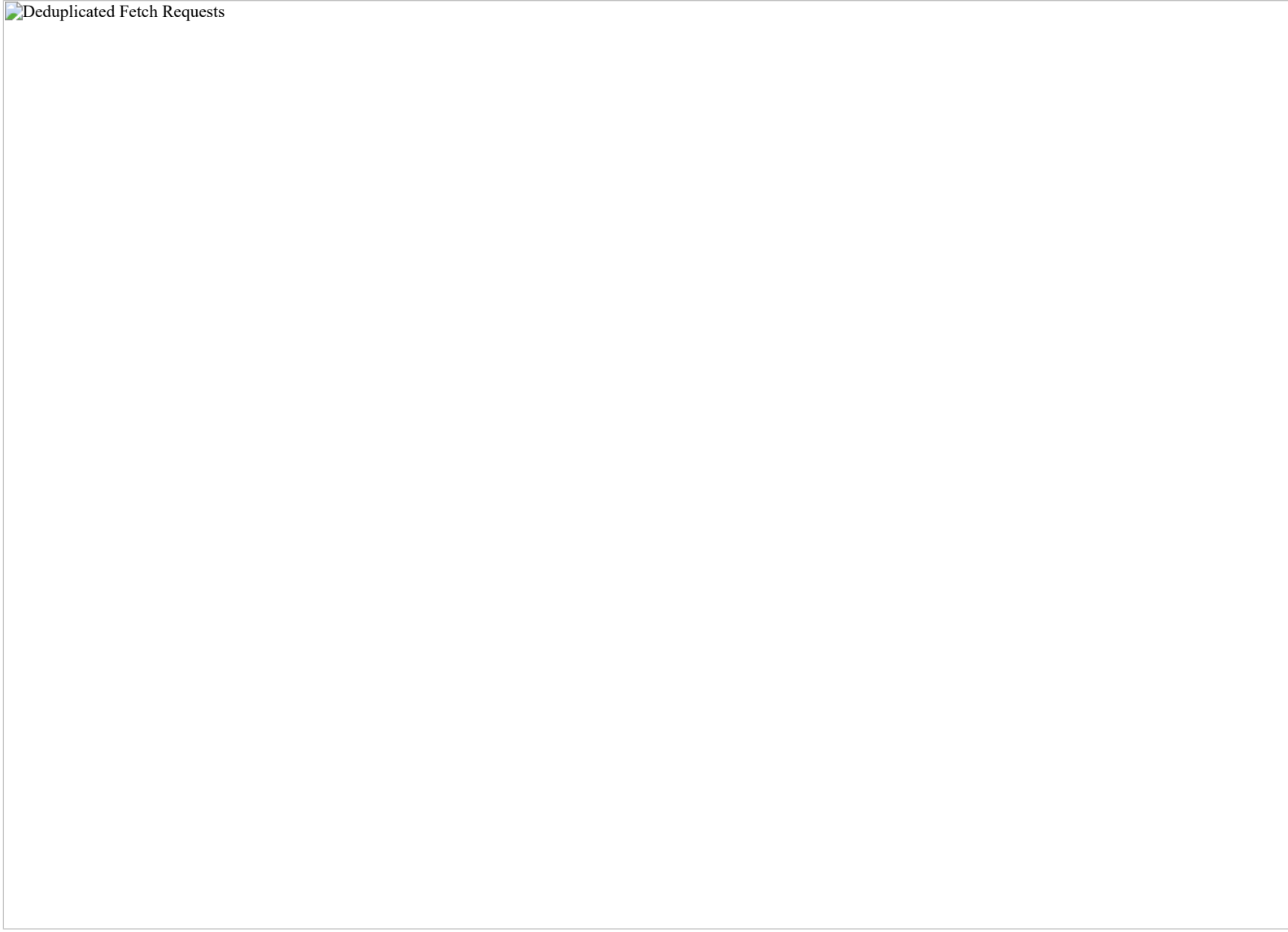
By default, Next.js will cache as much as possible to improve performance and reduce cost. This means routes are **statically rendered** and data requests are **cached** unless you opt out. The diagram below shows the default caching behavior: when a route is statically rendered at build time and when 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, and 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

Memoization only applies to the `GET` method in `fetch` requests, other methods, such as `POST` and `DELETE`, are not memoized. This default behavior is a React optimization and we do not recommend opting out of it.

To manage individual requests, you can use the `signal` property from `AbortController`. However, this will not opt requests out of memoization, rather, abort in-flight requests.

```
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.

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 with the `'force-cache'` option 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. no `cache` option defined or using `{ 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.

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

If you do *not* want to cache the response from `fetch`, you can do the following:

```
let data = await fetch('https://api.vercel.app/blog', { cache: 'no-store' })
```

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 API**: 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.

Client-side Router Cache

Next.js has an in-memory client-side router cache that stores the RSC payload of route segments, split by layouts, loading states, and pages.

When a user navigates between routes, Next.js caches the visited route segments and prefetches the routes the user is likely to navigate to. This results in instant back/forward navigation, no full-page reload between navigations, and preservation of React state and browser state.

With the Router Cache:

- **Layouts** are cached and reused on navigation (partial rendering).
- **Loading states** are cached and reused on navigation for instant navigation.
- **Pages** are not cached by default, but are reused during browser backward and forward navigation. You can enable caching for page segments by using the experimental `staleTimes` config option.

{/* TODO: Update diagram to match v15 behavior */}

> **Good to know:** This cache specifically applies to Next.js and Server Components, and is different to the browser's bfcache, though it has a similar result.

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 layouts and loading states is automatically invalidated after a specific time. The duration depends on how the resource was prefetched, and if the resource was statically generated:
  - **Default Prefetching** (`prefetch={null}` or unspecified): not cached for dynamic pages, 5 minutes for static pages.
  - **Full Prefetching** (`prefetch={true}` or `router.prefetch`): 5 minutes for both static & dynamic pages.

While a page refresh will clear **all** cached segments, the automatic invalidation period only affects the individual segment from the time it was prefetched.

> **Good to know**: The experimental `staleTimes` config option can be used to adjust the automatic invalidation times mentioned above.

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

As of Next.js 15, page segments are opted out by default.

> **Good to know:** You can also opt out of prefetching by setting the `prefetch` prop of the `<Link>` component to `false`.

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

- To immediately invalidate the Data Cache and Router cache, you can use `revalidatePath` or `revalidateTag` in a Server Action.
- 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.

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		

API	Router Cache	Full Route Cache	Data Cache	React Cache
`headers`, `searchParams`		Opt out		
`generateStaticParams`		Cache		
`React.cache`				Cache
`unstable_cache`			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.

If you do *not* want to cache the response from `fetch`, you can do the following:

```
let data = await fetch('https://api.vercel.app/blog', { cache: 'no-store' })
```

See the `fetch` API Reference for more options.

`fetch options.cache`

You can opt individual `fetch` into caching by setting the `cache` option to `force-cache`:

```
// Opt into caching
fetch(`https://...`, { cache: 'force-cache' })
```

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 APIs

Dynamic APIs 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 Full Route Cache:

- `const dynamic = 'force-dynamic'`

This config option will opt all fetches out of the Data Cache (i.e. `no-store`):

- `const fetchCache = 'default-no-store'`

See the `fetchCache` to see more advanced options.

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.

To statically render all paths at build time, supply the full list of paths to `generateStaticParams`:

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

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

To statically render a subset of paths at build time, and the rest the first time they're visited at runtime, return a partial list of paths:

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

  // Render the first 10 posts at build time
  return posts.slice(0, 10).map((post) => ({
    slug: post.slug,
  }))
}
```

To statically render all paths the first time they're visited, return an empty array (no paths will be rendered at build time) or utilize `export const dynamic = 'force-static'`:

```
export async function generateStaticParams() {
  return []
}
```

> **Good to know:** You must return an array from `generateStaticParams`, even if it's empty. Otherwise, the route will be dynamically rendered.

```
export const dynamic = 'force-static'
```

To disable caching at request time, add the `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).

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 nav_title: CSS description: Style your Next.js Application with CSS Modules, Global Styles, and external stylesheets.

{/* 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 supports multiple ways of handling CSS, including:

- CSS Modules
- Global Styles
- External Stylesheets

CSS Modules

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

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

Example

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

```
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 **only enabled for files with the `.module.css` and `.module.sass` extensions**.

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.
> - Next.js does not support usage of global styles unless they are actually global, meaning they can apply to all pages and can live for the lifetime of the application. This is because Next.js uses React's built-in support for stylesheets to integrate with Suspense. This built-in support currently does not remove stylesheets as you navigate between routes. Because of this, we recommend using CSS Modules over global styles.

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>
  )
}
```

Ordering and Merging

Next.js optimizes CSS during production builds by automatically chunking (merging) stylesheets. The CSS order is *determined by the order in which you import the stylesheets into your application code*.

For example, `base-button.module.css` will be ordered before `page.module.css` since `<BaseButton>` is imported first in `<Page>`:

```
import styles from './base-button.module.css'

export function BaseButton() {
  return <button className={styles.primary} />
}
```

```
import styles from './base-button.module.css'

export function BaseButton() {
  return <button className={styles.primary} />
}
```

```
import { BaseButton } from './base-button'
import styles from './page.module.css'

export function Page() {
  return <BaseButton className={styles.primary} />
}
```

```
import { BaseButton } from './base-button'
import styles from './page.module.css'

export function Page() {
  return <BaseButton className={styles.primary} />
}
```

To maintain a predictable order, we recommend the following:

- Only import a CSS file in a single JS/TS file.
  - If using global class names, import the global styles in the same file in the order you want them to be applied.
- Prefer CSS Modules over global styles.
  - Use a consistent naming convention for your CSS modules. For example, using `<name>.module.css` over `<name>.tsx`.
- Extract shared styles into a separate shared component.
- If using Tailwind, import the stylesheet at the top of the file, preferably in the Root Layout.
- Turn off any linters/formatters (e.g., ESLint's `sort-import`) that automatically sort your imports. This can inadvertently affect your CSS since CSS import order *matters*.

  Good to know:

  - CSS ordering can behave differently in development mode, always ensure to check the build (`next build`) to verify the final CSS order.
  - You can use the `cssChunking` option in `next.config.js` to control how CSS is chunked.

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 your Tailwind configuration file, add paths to the files that will use Tailwind class names:

```
import type { Config } from 'tailwindcss'

const config: Config = {
  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: [],
}
export default config
```

```
/** @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: 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 your Sass options, use `sassOptions` in `next.config`.

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

const nextConfig: NextConfig = {
  sassOptions: {
    additionalData: `$var: red;`,
  },
}

export default nextConfig
```

```
/** @type {import('next').NextConfig} */

const nextConfig = {
  sassOptions: {
    additionalData: `$var: red;`,
  },
}

module.exports = nextConfig
```

Implementation

You can use the `implementation` property to specify the Sass implementation to use. By default, Next.js uses the `sass` package.

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

const nextConfig: NextConfig = {
  sassOptions: {
    implementation: 'sass-embedded',
  },
}

export default nextConfig
```

```
/** @type {import('next').NextConfig} */
```

```
const nextConfig = {
  sassOptions: {
    implementation: 'sass-embedded',
  },
}

module.exports = nextConfig
```

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: 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):

- `ant-design`
- `chakra-ui`
- `@fluentui/react-components`
- `kuma-ui`
- `@mui/material`
- `@mui/joy`
- `pandacss`
- `styled-jsx`
- `styled-components`
- `stylex`
- `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: 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:

- **CSS Modules**: Create locally scoped CSS classes to avoid naming conflicts and improve maintainability.
- **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.
- **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 intrinsic `width` and `height` of your image based on the imported file. These values are used to determine the image ratio and 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.

You can optionally configure `localPatterns` in your `next.config.js` file in order to allow specific images and block all others.

```
module.exports = {
  images: {
    localPatterns: [
      {
        pathname: '/assets/images/**',
```

```
        search: '',
      },
    ],
  },
}
```

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. Manually, by including a [`width`](#) and [`height`](#) property used to determine the image's aspect ratio.
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.

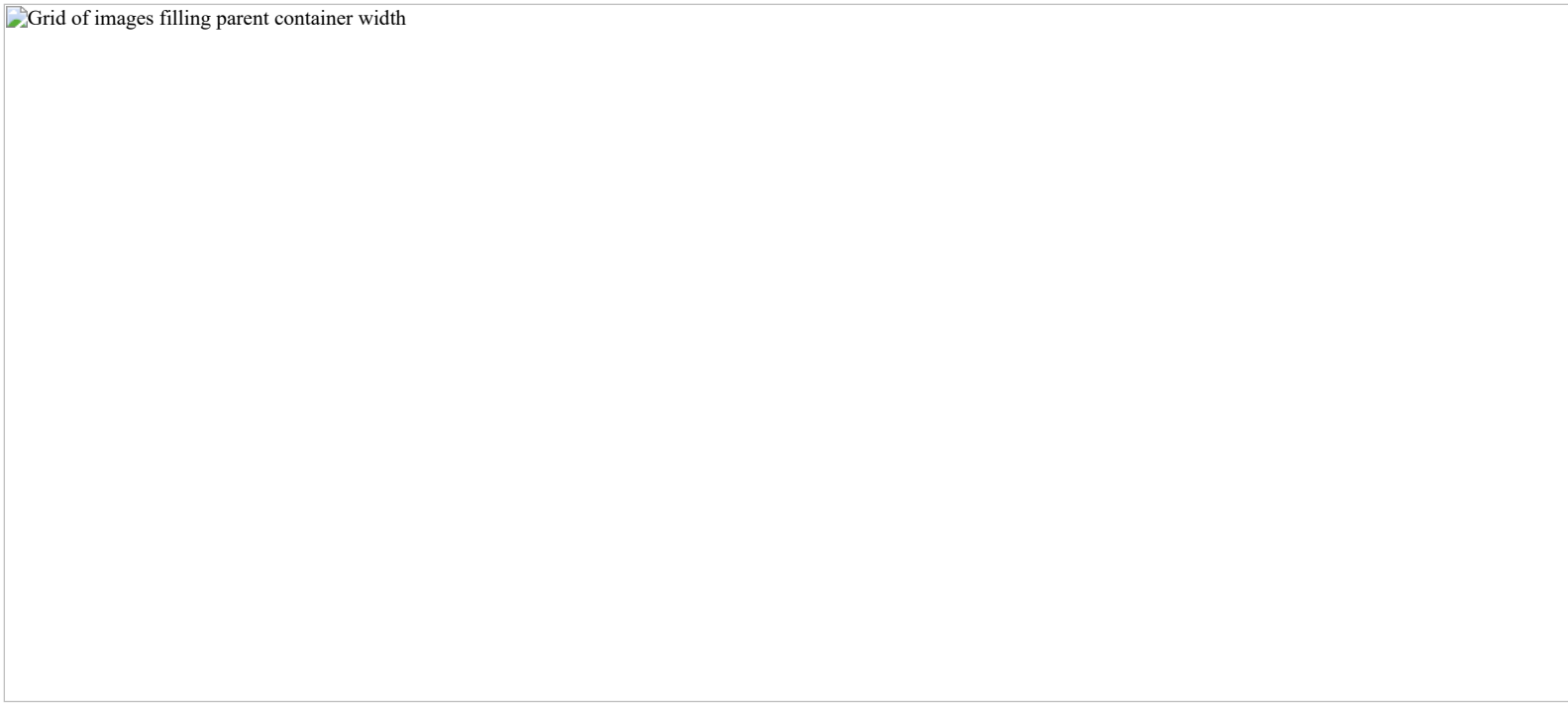
Examples

Responsive


```
import Image from 'next/image'
import mountains from '../public/mountains.jpg'

export default function Responsive() {
  return (
    <div style={{ display: 'flex', flexDirection: 'column' }}>
```

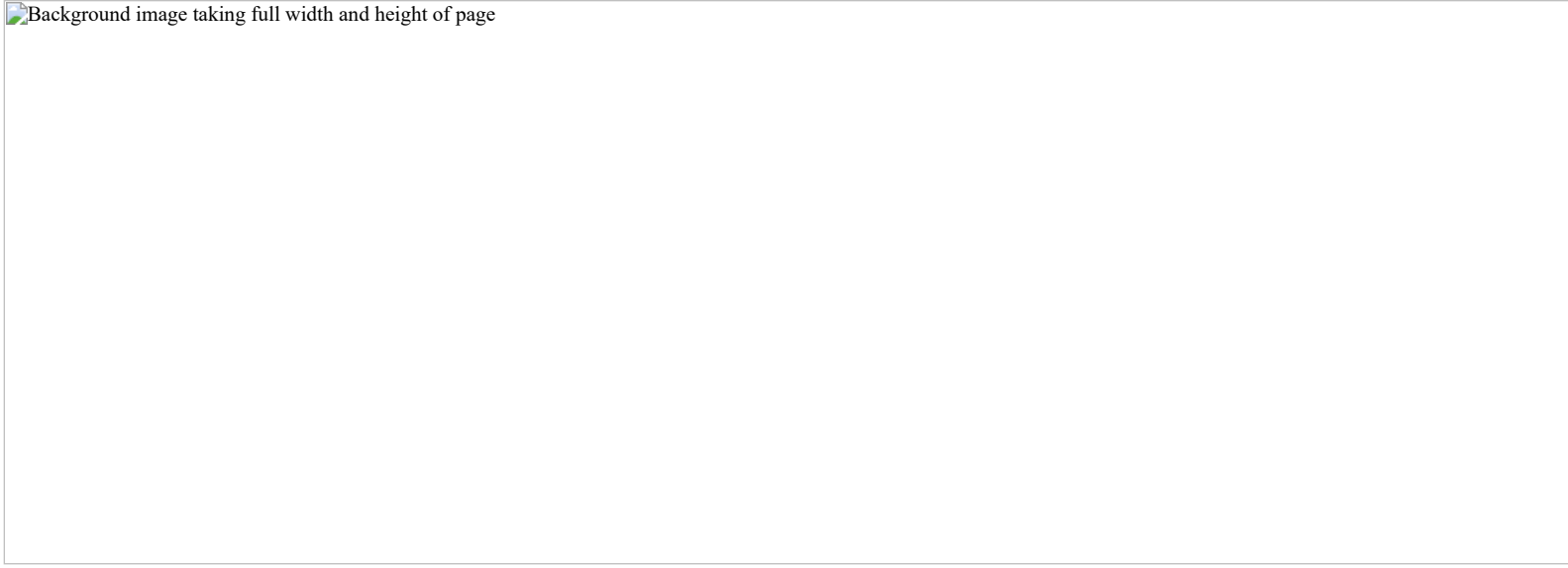
```
      <Image
        alt="Mountains"
        // Importing an image will
        // automatically set the width and height
        src={mountains}
        sizes="100vw"
        // Make the image display full width
        style={{
          width: '100%',
          height: 'auto',
        }}
      />
    </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: Video Optimization nav_title: Videos description: Recommendations and best practices for optimizing videos in your Next.js application.

This page outlines how to use videos with Next.js applications, showing how to store and display video files without affecting performance.

Using `<video>` and `<iframe>`

Videos can be embedded on the page using the HTML **`<video>`** tag for direct video files and **`<iframe>`** for external platform-hosted videos.

`<video>`

The HTML `<video>` tag can embed self-hosted or directly served video content, allowing full control over the playback and appearance.

```
export function Video() {
  return (
    <video width="320" height="240" controls preload="none">
      <source src="/path/to/video.mp4" type="video/mp4" />
      <track
        src="/path/to/captions.vtt"
        kind="subtitles"
        srcLang="en"
        label="English"
      />
      Your browser does not support the video tag.
    </video>
  )
}
```

Common `<video>` tag attributes

Attribute	Description	Example Value
`src`	Specifies the source of the video file.	`<video src="/path/to/video.mp4" />`
`width`	Sets the width of the video player.	`<video width="320" />`
`height`	Sets the height of the video player.	`<video height="240" />`
`controls`	If present, it displays the default set of playback controls.	`<video controls />`
`autoPlay`	Automatically starts playing the video when the page loads. Note: Autoplay policies vary across browsers.	`<video autoPlay />`

Attribute	Description	Example Value
`loop`	Loops the video playback.	`<video loop />`
`muted`	Mutes the audio by default. Often used with `autoPlay`.	`<video muted />`
`preload`	Specifies how the video is preloaded. Values: `none`, `metadata`, `auto`.	`<video preload="none" />`
`playsInline`	Enables inline playback on iOS devices, often necessary for autoplay to work on iOS Safari.	`<video playsInline />`

> **Good to know**: When using the `autoPlay` attribute, it is important to also include the `muted` attribute to ensure the video plays automatically in most browsers and the `playsInline` attribute for compatibility with iOS devices.

For a comprehensive list of video attributes, refer to the MDN documentation.

Video best practices

- **Fallback Content:** When using the `<video>` tag, include fallback content inside the tag for browsers that do not support video playback.
- **Subtitles or Captions:** Include subtitles or captions for users who are deaf or hard of hearing. Utilize the `<track>` tag with your `<video>` elements to specify caption file sources.
- **Accessible Controls:** Standard HTML5 video controls are recommended for keyboard navigation and screen reader compatibility. For advanced needs, consider third-party players like react-player or video.js, which offer accessible controls and consistent browser experience.

`<iframe>`

The HTML `<iframe>` tag allows you to embed videos from external platforms like YouTube or Vimeo.

```
export default function Page() {
  return (
    <iframe
      src="https://www.youtube.com/watch?v=gfU1iZnjRZM"
      frameborder="0"
      allowfullscreen
    />
  )
}
```

Common `<iframe>` tag attributes

Attribute	Description	Example Value
`src`	The URL of the page to embed.	`<iframe src="https://example.com" />`
`width`	Sets the width of the iframe.	`<iframe width="500" />`
`height`	Sets the height of the iframe.	`<iframe height="300" />`
`frameborder`	Specifies whether or not to display a border around the iframe.	`<iframe frameborder="0" />`
`allowfullscreen`	Allows the iframe content to be displayed in full-screen mode.	`<iframe allowfullscreen />`
`sandbox`	Enables an extra set of restrictions on the content within the iframe.	`<iframe sandbox />`
`loading`	Optimize loading behavior (e.g., lazy loading).	`<iframe loading="lazy" />`
`title`	Provides a title for the iframe to support accessibility.	`<iframe title="Description" />`

For a comprehensive list of iframe attributes, refer to the MDN documentation.

Choosing a video embedding method

There are two ways you can embed videos in your Next.js application:

- **Self-hosted or direct video files:** Embed self-hosted videos using the `<video>` tag for scenarios requiring detailed control over the player's functionality and appearance. This integration method within Next.js allows for customization and control of your video content.
- **Using video hosting services (YouTube, Vimeo, etc.):** For video hosting services like YouTube or Vimeo, you'll embed their iframe-based players using the `<iframe>` tag. While this method limits some control over the player, it offers ease of use and features provided by these platforms.

Choose the embedding method that aligns with your application's requirements and the user experience you aim to deliver.

Embedding externally hosted videos

To embed videos from external platforms, you can use Next.js to fetch the video information and React Suspense to handle the fallback state while loading.

1. Create a Server Component for video embedding

The first step is to create a Server Component that generates the appropriate iframe for embedding the video. This component will fetch the source URL for the video and render the iframe.

```
export default async function VideoComponent() {
  const src = await getVideoSrc()

  return <iframe src={src} frameborder="0" allowfullscreen />
}
```

2. Stream the video component using React Suspense

After creating the Server Component to embed the video, the next step is to stream the component using React Suspense.

```
import { Suspense } from 'react'
import VideoComponent from '../ui/VideoComponent.jsx'

export default function Page() {
  return (
    <section>
      <Suspense fallback={<p>Loading video...</p>}>
        <VideoComponent />
      </Suspense>
      {/* Other content of the page */}
    </section>
  )
}
```

> **Good to know**: When embedding videos from external platforms, consider the following best practices:
>
> - Ensure the video embeds are responsive. Use CSS to make the iframe or video player adapt to different screen sizes.
> - Implement strategies for loading videos based on network conditions, especially for users with limited data plans.

This approach results in a better user experience as it prevents the page from blocking, meaning the user can interact with the page while the video component streams in.

For a more engaging and informative loading experience, consider using a loading skeleton as the fallback UI. So instead of showing a simple loading message, you can show a skeleton that resembles the video player like this:

```
import { Suspense } from 'react'
import VideoComponent from '../ui/VideoComponent.jsx'
import VideoSkeleton from '../ui/VideoSkeleton.jsx'

export default function Page() {
  return (
    <section>
      <Suspense fallback={<VideoSkeleton />}>
        <VideoComponent />
      </Suspense>
      {/* Other content of the page */}
    </section>
  )
}
```

Self-hosted videos

Self-hosting videos may be preferable for several reasons:

- **Complete control and independence**: Self-hosting gives you direct management over your video content, from playback to appearance, ensuring full ownership and control, free from external platform constraints.
- **Customization for specific needs**: Ideal for unique requirements, like dynamic background videos, it allows for tailored customization to align with design and functional needs.
- **Performance and scalability considerations**: Choose storage solutions that are both high-performing and scalable, to support increasing traffic and content size effectively.
- **Cost and integration**: Balance the costs of storage and bandwidth with the need for easy integration into your Next.js framework and broader tech ecosystem.

Using Vercel Blob for video hosting

Vercel Blob offers an efficient way to host videos, providing a scalable cloud storage solution that works well with Next.js. Here's how you can host a video using Vercel Blob:

1. Uploading a video to Vercel Blob

In your Vercel dashboard, navigate to the "Storage" tab and select your Vercel Blob store. In the Blob table's upper-right corner, find and click the "Upload" button. Then, choose the video file you wish to upload. After the upload completes, the video file will appear in the Blob table.

Alternatively, you can upload your video using a server action. For detailed instructions, refer to the Vercel documentation on server-side uploads. Vercel also supports client-side uploads. This method may be preferable for certain use cases.

2. Displaying the video in Next.js

Once the video is uploaded and stored, you can display it in your Next.js application. Here's an example of how to do this using the `<video>` tag and React Suspense:

```
import { Suspense } from 'react'
import { list } from '@vercel/blob'

export default function Page() {
  return (
    <Suspense fallback={<p>Loading video...</p>}>
      <VideoComponent fileName="my-video.mp4" />
    </Suspense>
  )
}

async function VideoComponent({ fileName }) {
  const { blobs } = await list({
    prefix: fileName,
    limit: 1,
  })
  const { url } = blobs[0]

  return (
    <video controls preload="none" aria-label="Video player">
      <source src={url} type="video/mp4" />
      Your browser does not support the video tag.
    </video>
  )
}
```

In this approach, the page uses the video's `@vercel/blob` URL to display the video using the `VideoComponent`. React Suspense is used to show a fallback until the video URL is fetched and the video is ready to be displayed.

Adding subtitles to your video

If you have subtitles for your video, you can easily add them using the `<track>` element inside your `<video>` tag. You can fetch the subtitle file from Vercel Blob in a similar way as the video file. Here's how you can update the `<VideoComponent>` to include subtitles.

```
async function VideoComponent({ fileName }) {
  const { blobs } = await list({
    prefix: fileName,
    limit: 2,
  })
  const { url } = blobs[0]
  const { url: captionsUrl } = blobs[1]

  return (
    <video controls preload="none" aria-label="Video player">
      <source src={url} type="video/mp4" />
      <track src={captionsUrl} kind="subtitles" srcLang="en" label="English" />
      Your browser does not support the video tag.
    </video>
  )
}
```

By following this approach, you can effectively self-host and integrate videos into your Next.js applications.

Resources

To continue learning more about video optimization and best practices, please refer to the following resources:

- **Understanding video formats and codecs**: Choose the right format and codec, like MP4 for compatibility or WebM for web optimization, for your video needs. For more details, see Mozilla's guide on video codecs.
- **Video compression**: Use tools like FFmpeg to effectively compress videos, balancing quality with file size. Learn about compression techniques at FFmpeg's official website.
- **Resolution and bitrate adjustment**: Adjust resolution and bitrate based on the viewing platform, with lower settings for mobile devices.
- **Content Delivery Networks (CDNs)**: Utilize a CDN to enhance video delivery speed and manage high traffic. When using some storage solutions, such as Vercel Blob, CDN functionality is automatically handled for you. Learn more about CDNs and their benefits.

Explore these video streaming platforms for integrating video into your Next.js projects:

Open source `next-video` component

- Provides a `<Video>` component for Next.js, compatible with various hosting services including Vercel Blob, S3, Backblaze, and Mux.
- Detailed documentation for using `next-video.dev` with different hosting services.

Cloudinary Integration

- Official documentation and integration guide for using Cloudinary with Next.js.
- Includes a `<CldVideoPlayer>` component for drop-in video support.
- Find examples of integrating Cloudinary with Next.js including Adaptive Bitrate Streaming.
- Other Cloudinary libraries including a Node.js SDK are also available.

Mux Video API

- Mux provides a starter template for creating a video course with Mux and Next.js.
- Learn about Mux's recommendations for embedding high-performance video for your Next.js application.
- Explore an example project demonstrating Mux with Next.js.

Fastly

- Learn more about integrating Fastly's solutions for video on demand and streaming media into Next.js.

ImageKit.io Integration

- Check out the official quick start guide for integrating ImageKit with Next.js.
- The integration provides an `<IKVideo>` component, offering seamless video support.
- You can also explore other ImageKit libraries, such as the Node.js SDK, which is also available.

---

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 using `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: 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: Promise<{ id: string }>
  searchParams: Promise<{ [key: string]: string | string[] | undefined }>
}

export async function generateMetadata(
  { params, searchParams }: Props,
  parent: ResolvingMetadata
): Promise<Metadata> {
  // read route params
  const id = (await 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 = (await 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.

To use it, you can import `ImageResponse` from `next/og`:

```
import { ImageResponse } from 'next/og'

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: 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.

Using custom Partytown configuration

Although the `worker` strategy does not require any additional configuration to work, Partytown supports the use of a config object to modify some of its settings, including enabling `debug` mode and forwarding events and triggers.

If you would like to add additional configuration options, you can include it within the `<Head />` component used in a custom `_document.js`:

```
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html>
      <Head>
        <script
          data-partytown-config
          dangerouslySetInnerHTML={{
            __html: `
              partytown = {
                lib: "/_next/static/~partytown/",
                debug: true
              };
            `,
          }}
        />
      </Head>
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

In order to modify Partytown's configuration, the following conditions must be met:

1. The `data-partytown-config` attribute must be used in order to overwrite the default configuration used by Next.js
2. Unless you decide to save Partytown's library files in a separate directory, the `lib: "/_next/static/~partytown/"` property and value must be included in the configuration object in order to let Partytown know where Next.js stores the necessary static files.

   Note: If you are using an asset prefix and would like to modify Partytown's default configuration, you must include it as part of the `lib` path.

Take a look at Partytown's configuration options to see the full list of other properties that can be added.

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: Optimizing Package Bundling nav_title: Package Bundling description: Learn how to optimize your application's server and client bundles. related: description: Learn more about optimizing your application for production. links: - app/building-your-application/deploying/production-checklist

Bundling external packages can significantly improve the performance of your application. By default, packages imported inside Server Components and Route Handlers are automatically bundled by Next.js. This page will guide you through how to analyze and further optimize package bundling. By default, packages imported into your application are not bundled. This can impact performance or might not work if external packages are not pre-bundled, for example, if imported from a monorepo or `node_modules`. This page will guide you through how to analyze and configure package bundling.

Analyzing JavaScript bundles

`@next/bundle-analyzer` is a plugin for Next.js that helps you manage the size of your application bundles. It generates a visual report of the size of each package and their dependencies. You can use the information to remove large dependencies, split, or lazy-load your code.

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`.

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

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

module.exports = withBundleAnalyzer(nextConfig)
```

Generating a report

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. Periodically evaluating your application's bundles can help you maintain application performance over time.

Optimizing package imports

Some packages, such as icon libraries, can export hundreds of modules, which can cause performance issues in development and production.

You can optimize how these packages are imported by adding the `optimizePackageImports` option to your `next.config.js`. This option will only load the modules you *actually* use, while still giving you the convenience of writing import statements with many named exports.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    optimizePackageImports: ['icon-library'],
  },
}

module.exports = nextConfig
```

Next.js also optimizes some libraries automatically, thus they do not need to be included in the optimizePackageImports list. See the full list.

Bundling specific packages

To bundle specific packages, you can use the `transpilePackages` option in your `next.config.js`. This option is useful for bundling external packages that are not pre-bundled, for example, in a monorepo or imported from `node_modules`.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  transpilePackages: ['package-name'],
}

module.exports = nextConfig
```

Bundling all packages

To automatically bundle all packages (default behavior in the App Router), you can use the `bundlePagesRouterDependencies` option in your `next.config.js`.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  bundlePagesRouterDependencies: true,
}

module.exports = nextConfig
```

Opting specific packages out of bundling

If you have the `bundlePagesRouterDependencies` option enabled, you can opt specific packages out of automatic bundling using the `serverExternalPackages` option in your `next.config.js`:

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  // Automatically bundle external packages in the Pages Router:
  bundlePagesRouterDependencies: true,
  // Opt specific packages out of bundling for both App and Pages Router:
  serverExternalPackages: ['package-name'],
}

module.exports = nextConfig
```

Opting specific packages out of bundling

Since packages imported inside Server Components and Route Handlers are automatically bundled by Next.js, you can opt specific packages out of bundling using the `serverExternalPackages` option in your `next.config.js`.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  serverExternalPackages: ['package-name'],
}

module.exports = nextConfig
```

Next.js includes a list of popular packages that currently are working on compatibility and automatically opt-ed out. See the full list.

---

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.

> **Note:** `ssr: false` option will only work for client components, move it into client components ensure the client code-splitting working properly.

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. It will also help preload the static assets such as CSS when you're using it in Server Components.

```
import dynamic from 'next/dynamic'

// Server Component:
const ServerComponent = dynamic(() => import('../components/ServerComponent'))

export default function ServerComponentExample() {
  return (
    <div>
      <ServerComponent />
    </div>
  )
}
```

> **Note:** `ssr: false` option is not supported in Server Components. You will see an error if you try to use it in Server Components. `ssr: false` is not allowed with `next/dynamic` in Server Components. Please move it into a client component.

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`.

```
'use client'

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
>   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 related: title: Learn more about Instrumentation links: - app/api-reference/file-conventions/instrumentation

{/* 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. */}

Instrumentation is the process of using code to integrate monitoring and logging tools into your application. This allows you to track the performance and behavior of your application, and to debug issues in production.

Convention

To set up instrumentation, create `instrumentation.ts|js` file in the **root directory** of your project (or inside the `src` folder if using one).

Then, export a `register` function in the file. This function will be called **once** when a new Next.js server instance is initiated.

For example, to use Next.js with OpenTelemetry and @vercel/otel:

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

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

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

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

See the Next.js with OpenTelemetry example for a complete implementation.

> **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.

Examples

Importing files with side effects

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.

We recommend importing files using JavaScript `import` syntax within your `register` function. 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')
}
```

Good to know:

We recommend importing the file from within the `register` function, rather than at the top of the file. By doing this, you can colocate all of your side effects in one place in your code, and avoid any unintended consequences from importing globally at the top of the file.

Importing runtime-specific code

Next.js calls `register` in all environments, so it's important to conditionally import any code that doesn't support specific runtimes (e.g. Edge or Node.js). You can use the `NEXT_RUNTIME` environment variable to get the current environment:

```
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. */}

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.

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.

Using `@vercel/otel`

To get started, install the following packages:

```
npm install @vercel/otel @opentelemetry/sdk-logs @opentelemetry/api-logs @opentelemetry/instrumentation
```

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({ serviceName: 'next-app' })
}
```


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

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


See the `@vercel/otel` documentation for additional configuration options.

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

The `@vercel/otel` package provides many configuration options and should serve most of common use cases. But if it 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-trace-otlp-http
```

Now you can initialize `NodeSDK` in your `instrumentation.ts`. Unlike `@vercel/otel`, `NodeSDK` is 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 { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http'
import { Resource } from '@opentelemetry/resources'
import { NodeSDK } from '@opentelemetry/sdk-node'
import { SimpleSpanProcessor } from '@opentelemetry/sdk-trace-node'
import { ATTR_SERVICE_NAME } from '@opentelemetry/semantic-conventions'

const sdk = new NodeSDK({
  resource: new Resource({
    [ATTR_SERVICE_NAME]: 'next-app',
  }),
  spanProcessor: new SimpleSpanProcessor(new OTLPTraceExporter()),
})
sdk.start()
```

```
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http'
import { Resource } from '@opentelemetry/resources'
import { NodeSDK } from '@opentelemetry/sdk-node'
import { SimpleSpanProcessor } from '@opentelemetry/sdk-trace-node'
import { ATTR_SERVICE_NAME } from '@opentelemetry/semantic-conventions'

const sdk = new NodeSDK({
  resource: new Resource({
    [ATTR_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 some features that are not exposed by the `@vercel/otel`. If edge runtime support is necessary, you will have to use `@vercel/otel`.

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

OpenTelemetry Collector is not necessary. You can use a custom OpenTelemetry exporter with `@vercel/otel` or 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.rsc` (true/false) - Whether the request is an RSC request, such as prefetch.
- `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`

This span can be turned off by setting `NEXT_OTEL_FETCH_DISABLED=1` in your environment. This is useful when you want to use a custom fetch instrumentation library.

`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`

`resolve page components`

- `next.span_type`: `NextNodeServer.findPageComponents`.

This span represents the process of resolving page components for a specific page.

Attributes:

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

`resolve segment modules`

- `next.span_type`: `NextNodeServer.getLayoutOrPageModule`.

This span represents loading of code modules for a layout or a page.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.segment`

`start response`

- `next.span_type`: `NextNodeServer.startResponse`.

This zero-length span represents the time when the first byte has been sent in the response.

---

title: Static 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, the file `public/avatars/me.png` can be viewed by visiting the `/avatars/me.png` path. The code to display that image might look like:

```
import Image from 'next/image'

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

export function AvatarOfMe() {
  return <Avatar id="me" alt="A portrait of me" />
}
```

Caching

Next.js cannot safely cache assets in the `public` folder because they may change. The default caching headers applied are:

```
Cache-Control: public, max-age=0
```

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 Vercel Blob for persistent file storage.

---

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.

All third-party integrations provided by `@next/third-parties` have been optimized for performance and ease of use.

Getting Started

To get started, install the `@next/third-parties` library:

```
npm install @next/third-parties@latest next@latest
```

{/* To do: Remove this paragraph once package becomes stable */}

`@next/third-parties` is currently an **experimental** library under active development. We recommend installing it with the **latest** or **canary** flags while we work on adding more third-party integrations.

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">
      <GoogleTagManager gtmId="GTM-XYZ" />
      <body>{children}</body>
    </html>
  )
}
```

```
import { GoogleTagManager } from '@next/third-parties/google'

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <GoogleTagManager gtmId="GTM-XYZ" />
      <body>{children}</body>
    </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.

Server-side Tagging

If you're using a server-side tag manager and serving `gtm.js` scripts from your tagging server you can use `gtmScriptUrl` option to specify the URL of the script.

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-`.
`gtmScriptUrl`	Optional	GTM script URL. Defaults to `https://www.googletagmanager.com/gtm.js`.
`dataLayer`	Optional	Data layer object to instantiate the container with.
`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`.
`nonce`	Optional	A nonce.

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: Memory Usage description: Optimize memory used by your application in development and production.

As applications grow and become more feature rich, they can demand more resources when developing locally or creating production builds.

Let's explore some strategies and techniques to optimize memory and address common memory issues in Next.js.

Reduce number of dependencies

Applications with a large amount of dependencies will use more memory.

The Bundle Analyzer can help you investigate large dependencies in your application that may be able to be removed to improve performance and memory usage.

Try `experimental.webpackMemoryOptimizations`

Starting in `v15.0.0`, you can add `experimental.webpackMemoryOptimizations: true` to your `next.config.js` file to change behavior in Webpack that reduces max memory usage but may increase compilation times by a slight amount.

> **Good to know**: This feature is currently experimental to test on more projects first, but it is considered to be low-risk.

Run `next build` with `--experimental-debug-memory-usage`

Starting in `14.2.0`, you can run `next build --experimental-debug-memory-usage` to run the build in a mode where Next.js will print out information about memory usage continuously throughout the build, such as heap usage and garbage collection statistics. Heap snapshots will also be taken automatically when memory usage gets close to the configured limit.

> **Good to know**: This feature is not compatible with the Webpack build worker option which is auto-enabled unless you have custom webpack config.

Record a heap profile

To look for memory issues, you can record a heap profile from Node.js and load it in Chrome DevTools to identify potential sources of memory leaks.

In your terminal, pass the `--heap-prof` flag to Node.js when starting your Next.js build:

```
node --heap-prof node_modules/next/dist/bin/next build
```

At the end of the build, a `.heapprofile` file will be created by Node.js.

In Chrome DevTools, you can open the Memory tab and click on the "Load Profile" button to visualize the file.

Analyze a snapshot of the heap

You can use an inspector tool to analyze the memory usage of the application.

When running the `next build` or `next dev` command, add `NODE_OPTIONS=--inspect` to the beginning of the command. This will expose the inspector agent on the default port. If you wish to break before any user code starts, you can pass `--inspect-brk` instead. While the process is running, you can use a tool such as Chrome DevTools to connect to the debugging port to record and analyze a snapshot of the heap to see what memory is being retained.

Starting in `14.2.0`, you can also run `next build` with the `--experimental-debug-memory-usage` flag to make it easier to take heap snapshots.

While running in this mode, you can send a `SIGUSR2` signal to the process at any point, and the process will take a heap snapshot.

The heap snapshot will be saved to the project root of the Next.js application and can be loaded in any heap analyzer, such as Chrome DevTools, to see what memory is retained. This mode is not yet compatible with Webpack build workers.

See how to record and analyze heap snapshots for more information.

Webpack build worker

The Webpack build worker allows you to run Webpack compilations inside a separate Node.js worker which will decrease memory usage of your application during builds.

This option is enabled by default if your application does not have a custom Webpack configuration starting in `v14.1.0`.

If you are using an older version of Next.js or you have a custom Webpack configuration, you can enable this option by setting `experimental.webpackBuildWorker: true` inside your `next.config.js`.

> **Good to know**: This feature may not be compatible with all custom Webpack plugins.

Disable Webpack cache

The Webpack cache saves generated Webpack modules in memory and/or to disk to improve the speed of builds. This can help with performance, but it will also increase the memory usage of your application to store the cached data.

You can disable this behavior by adding a custom Webpack configuration to your application:

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  webpack: (
    config,
    { buildId, dev, isServer, defaultLoaders, nextRuntime, webpack }
  ) => {
    if (config.cache && !dev) {
      config.cache = Object.freeze({
        type: 'memory',
      })
    }
    // Important: return the modified config
    return config
  },
}

export default nextConfig
```

Disable static analysis

Typechecking and linting may require a lot of memory, especially in large projects. However, most projects have a dedicated CI runner that already handles these tasks. When the build produces out-of-memory issues during the "Linting and checking validity of types" step, you can disable these task during builds:

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  eslint: {
    // Warning: This allows production builds to successfully complete even if
    // your project has ESLint errors.
    ignoreDuringBuilds: true,
  },
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}

export default nextConfig
```

- Ignoring TypeScript Errors
- ESLint in Next.js config

Keep in mind that this may produce faulty deploys due to type errors or linting issues. We strongly recommend only promoting builds to production after static analysis has completed. If you deploy to Vercel, you can check out the guide for staging deployments to learn how to promote builds to production after custom tasks have succeeded.

Disable source maps

Generating source maps consumes extra memory during the build process.

You can disable source map generation by adding `productionBrowserSourceMaps: false` and `experimental.serverSourceMaps: false` to your Next.js configuration.

> **Good to know**: Some plugins may turn on source maps and may require custom configuration to disable.

Edge memory issues

Next.js `v14.1.3` fixed a memory issue when using the Edge runtime. Please update to this version (or later) to see if it addresses your issue.

---

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.

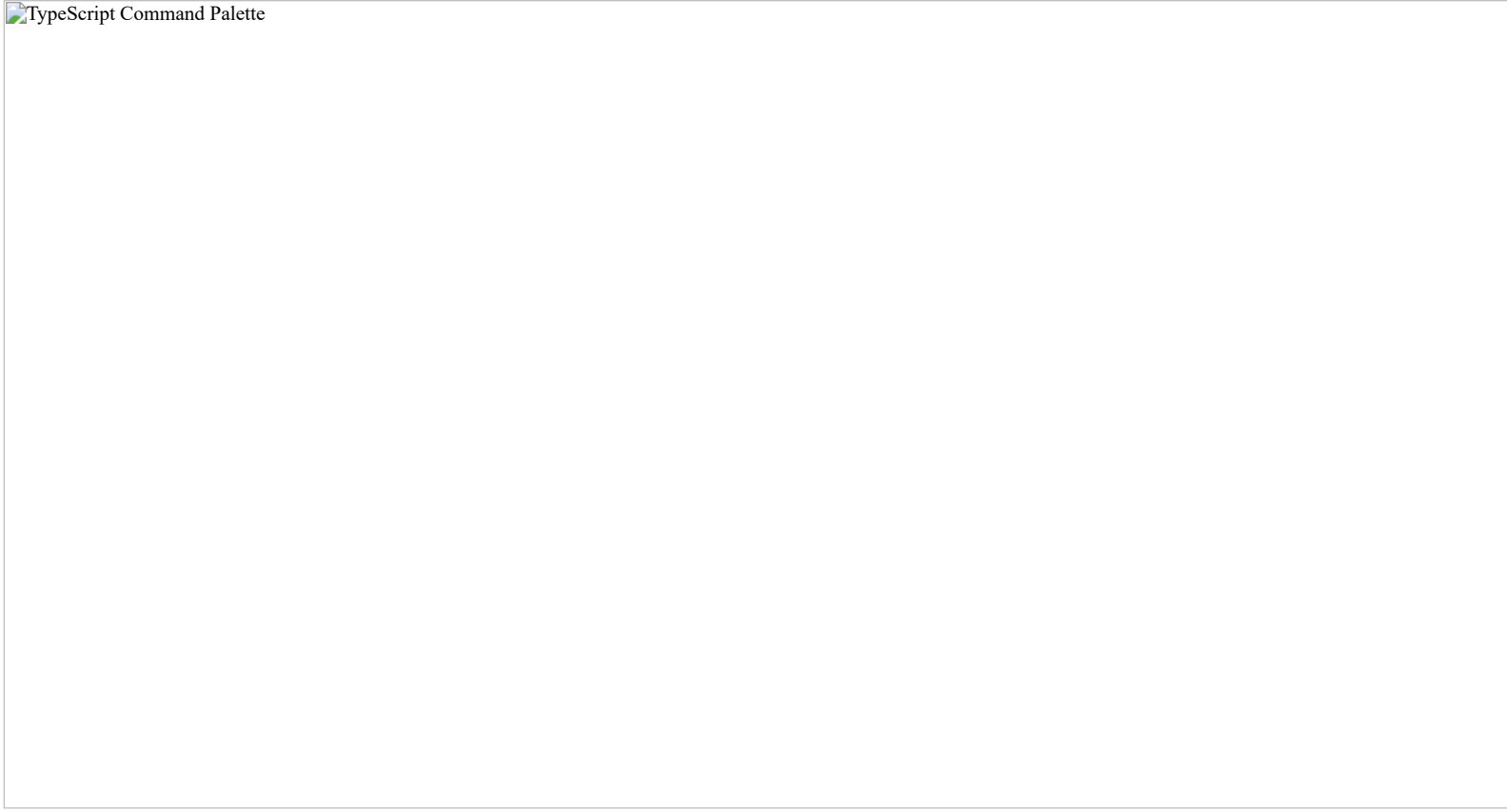
We also recommend you to use `next.config.ts` over `next.config.js` for better type inference.

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.

Type checking in Next.js Configuration

Type checking next.config.js

When using the `next.config.js` file, 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
```

Type checking next.config.ts

You can use TypeScript and import types in your Next.js configuration by using `next.config.ts`.

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

const nextConfig: NextConfig = {
  /* config options here */
}

export default nextConfig
```

> **Note**: Module resolution in `next.config.ts` is currently limited to `CommonJS`. This may cause incompatibilities with ESM only packages being loaded in `next.config.ts`.

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.

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

const nextConfig: NextConfig = {
  experimental: {
    typedRoutes: true,
  },
```

```
}

export default 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

The Next.js App Router 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 the App Router, 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 type { 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.

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.ts` and enable the `ignoreBuildErrors` option in the `typescript` config:

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

const nextConfig: NextConfig = {
  typescript: {
    // !! WARN !!
    // Dangerously allow production builds to successfully complete even if
    // your project has type errors.
    // !! WARN !!
    ignoreBuildErrors: true,
  },
}

export default nextConfig
```

> **Good to know**: You can run `tsc --noEmit` to check for TypeScript errors yourself before building. This is useful for CI/CD pipelines where you'd like to check for TypeScript errors before deploying.

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
`v15.0.0`	`next.config.ts` support added for TypeScript projects.
`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.

Rule	Description
@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.

TypeScript

In addition to the Next.js ESLint rules, `create-next-app --typescript` will also add TypeScript-specific lint rules with `next/typescript` to your config:

```
{
  "extends": ["next/core-web-vitals", "next/typescript"]
}
```

Those rules are based on `plugin:@typescript-eslint/recommended`. See typescript-eslint > Configs for more details.

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` 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*` files 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
>
> # you can write with line breaks
> PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----
> ...
> Kh9NV...
> ...
> -----END DSA PRIVATE KEY-----"
>
> # or with `\n` inside double quotes
> PRIVATE_KEY="-----BEGIN RSA PRIVATE KEY-----\nKh9NV...\n-----END DSA PRIVATE KEY-----\n"
> ```
>
> **Note**: If you are using a `/src` folder, please note that Next.js will load the .env files **only** from the parent folder and **not** from the `/src` folder. This loads `process.env.DB_HOST`, `process.env.DB_USER`, and `process.env.DB_PASS` into the Node.js environment automatically allowing you to use them in Route Handlers.

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

Loading Environment Variables with `@next/env`

If you need to load environment variables outside of the Next.js runtime, such as in a root config file for an ORM or test runner, you can use the `@next/env` package.

This package is used internally by Next.js to load environment variables from `.env*` files.

To use it, install the package and use the `loadEnvConfig` function to load the environment variables:

```
npm install @next/env
```

```
import { loadEnvConfig } from '@next/env'

const projectDir = process.cwd()
loadEnvConfig(projectDir)
```

```
import { loadEnvConfig } from '@next/env'

const projectDir = process.cwd()
loadEnvConfig(projectDir)
```

Then, you can import the configuration where needed. For example:

```
import './envConfig.ts'

export default defineConfig({
  dbCredentials: {
    connectionString: process.env.DATABASE_URL!,
  },
})
```

```
import './envConfig.js'

export default defineConfig({
  dbCredentials: {
    connectionString: process.env.DATABASE_URL,
```

```
  },
})
```

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://x.com/$TWITTER_USER
```

In the above example, `process.env.TWITTER_URL` would be set to `https://x.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.

You can safely read environment variables on the server during dynamic rendering:

```
import { connection } from 'next/server'

export default async function Component() {
  await connection()
  // cookies, headers, and other Dynamic APIs
  // will also opt into dynamic rendering, meaning
  // this env variable is evaluated at runtime
  const value = process.env.MY_VALUE
  // ...
}
```

```
import { connection } from 'next/server'

export default async function Component() {
  await connection()
  // cookies, headers, and other Dynamic APIs
  // will also opt into dynamic rendering, meaning
  // this env variable is evaluated at runtime
  const value = process.env.MY_VALUE
  // ...
}
```

This allows you to use a singular Docker image that can be promoted through multiple environments with different values.

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

Typically, only `.env*` 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).

> **Good to know**: `.env`, `.env.development`, and `.env.production` files should be included in your repository as they define defaults. All `.env` files are excluded in `.gitignore` by default, allowing you to opt-into committing these values to your repository.

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
```

> **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:

```
{
  "compilerOptions": {
    "baseUrl": "src/",
    "paths": {
      "@/styles/*": ["styles/*"],
      "@/components/*": ["components/*"]
    }
  }
}
```

```
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>
  )
}
```

```
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 and use it in your Next.js apps.

{/* 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).

> **Good to know**: View the Portfolio Starter Kit template for a complete working example.

Install dependencies

The `@next/mdx` package, and related packages, are 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 `.md` or `.mdx` extension, directly in your `/pages` or `/app` directory.

Install these packages to render MDX with Next.js:

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

Configure `next.config.mjs`

Update the `next.config.mjs` file at your project's root to configure it to use MDX:

```
import createMDX from '@next/mdx'

/** @type {import('next').NextConfig} */
const nextConfig = {
  // Configure `pageExtensions` to include markdown and MDX files
  pageExtensions: ['js', 'jsx', 'md', 'mdx', 'ts', 'tsx'],
  // Optionally, add any other Next.js config below
}

const withMDX = createMDX({
  // Add markdown plugins here, as desired
})

// Merge MDX config with Next.js config
export default withMDX(nextConfig)
```

This allows `.md` and `.mdx` files to act as pages, routes, or imports in your application.

Add an `mdx-components.tsx` file

Create an `mdx-components.tsx` (or `.js`) file in the root of your project to define global MDX Components. For example, at the same level as `pages` or `app`, or inside `src` if applicable.

```
import type { MDXComponents } from 'mdx/types'

export function useMDXComponents(components: MDXComponents): MDXComponents {
  return {
    ...components,
  }
}

export function useMDXComponents(components) {
  return {
    ...components,
  }
}
```

> **Good to know**:
>
> - `mdx-components.tsx` is **required** to use `@next/mdx` with App Router and will not work without it.
> - Learn more about the `mdx-components.tsx` file convention.
> - Learn how to use custom styles and components.

Rendering MDX

You can render MDX using Next.js's file based routing or by importing MDX files into other pages.

Using file based routing

When using file based routing, you can use MDX pages like any other page.

In App Router apps, that includes being able to use metadata.

Create a new MDX page within the `/app` directory:

```
my-project
├── app
│   └── mdx-page
│       └── page.(mdx/md)
├── mdx-components.(tsx/js)
└── package.json
```

Create a new MDX page within the `/pages` directory:

```
my-project
├── mdx-components.(tsx/js)
├── pages
│   └── mdx-page.(mdx/md)
└── package.json
```

You can use MDX in these files, and even import React components, directly inside your MDX page:

```
import { MyComponent } from 'my-component'

# 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 `/mdx-page` route should display your rendered MDX page.

Using imports

Create a new page within the `/app` directory and an MDX file wherever you'd like:

```
my-project
├── app
│   └── mdx-page
│       └── page.(tsx/js)
├── markdown
│   └── welcome.(mdx/md)
├── mdx-components.(tsx/js)
└── package.json
```

Create a new page within the `/pages` directory and an MDX file wherever you'd like:

```
my-project
├── pages
│   └── mdx-page.(tsx/js)
├── markdown
│   └── welcome.(mdx/md)
├── mdx-components.(tsx/js)
└── package.json
```

You can use MDX in these files, and even import React components, directly inside your MDX page:

```
import { MyComponent } from 'my-component'

# 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 />
```

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

```
import Welcome from '@/markdown/welcome.mdx'

export default function Page() {
  return <Welcome />
}
```

```
import Welcome from '@/markdown/welcome.mdx'

export default function Page() {
  return <Welcome />
}
```

```
import Welcome from '@/markdown/welcome.mdx'

export default function Page() {
  return <Welcome />
}
```

```
import Welcome from '@/markdown/welcome.mdx'

export default function Page() {
  return <Welcome />
}
```

Navigating to the `/mdx-page` route should display your rendered MDX page.

Using custom styles and components

Markdown, when rendered, maps to native HTML elements. For example, writing the following markdown:

```
## This is a heading

This is a list in markdown:

- One
- Two
- Three
```

Generates the following HTML:

```
<h2>This is a heading</h2>

<p>This is a list in markdown:</p>

<ul>
  <li>One</li>
  <li>Two</li>
  <li>Three</li>
</ul>
```

To style your markdown, you can provide custom components that map to the generated HTML elements. Styles and components can be implemented globally, locally, and with shared layouts.

Global styles and components

Adding styles and components in `mdx-components.tsx` will affect *all* MDX files in your application.

```
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={{ color: 'red', fontSize: '48px' }}>{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={{ color: 'red', fontSize: '48px' }}>{children}</h1>
    ),
    img: (props) => (
      <Image
        sizes="100vw"
        style={{ width: '100%', height: 'auto' }}
        {...props}
      />
    ),
    ...components,
  }
}
```

Local styles and components

You can apply local styles and components to specific pages by passing them into imported MDX components. These will merge with and override global styles and components.

```
import Welcome from '@/markdown/welcome.mdx'

function CustomH1({ children }) {
  return <h1 style={{ color: 'blue', fontSize: '100px' }}>{children}</h1>
}

const overrideComponents = {
  h1: CustomH1,
}

export default function Page() {
  return <Welcome components={overrideComponents} />
}
```

```
import Welcome from '@/markdown/welcome.mdx'

function CustomH1({ children }) {
  return <h1 style={{ color: 'blue', fontSize: '100px' }}>{children}</h1>
}

const overrideComponents = {
  h1: CustomH1,
}
```

```
export default function Page() {
  return <Welcome components={overrideComponents} />
}

import Welcome from '@/markdown/welcome.mdx'

function CustomH1({ children }) {
  return <h1 style={{ color: 'blue', fontSize: '100px' }}>{children}</h1>
}

const overrideComponents = {
  h1: CustomH1,
}

export default function Page() {
  return <Welcome components={overrideComponents} />
}
```


```
import Welcome from '@/markdown/welcome.mdx'

function CustomH1({ children }) {
  return <h1 style={{ color: 'blue', fontSize: '100px' }}>{children}</h1>
}

const overrideComponents = {
  h1: CustomH1,
}

export default function Page() {
  return <Welcome components={overrideComponents} />
}
```


Shared layouts

To share a layout across 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>

}
```

Using Tailwind typography plugin

If you are using Tailwind to style your application, using the `@tailwindcss/typography` plugin will allow you to reuse your Tailwind configuration and styles in your markdown files.

The plugin adds a set of `prose` classes that can be used to add typographic styles to content blocks that come from sources, like markdown.

Install Tailwind typography and use with shared layouts to add the `prose` you want.

```
export default function MdxLayout({ children }: { children: React.ReactNode }) {
  // Create any shared layout or styles here
  return (
    <div className="prose prose-headings:mt-8 prose-headings:font-semibold prose-headings:text-black prose-h1:text-5xl prose-h2:text-4xl prose-h3:text-3xl prose-h4:te
      {children}
    </div>
  )
}
```


```
export default function MdxLayout({ children }) {
  // Create any shared layout or styles here
  return (
    <div className="prose prose-headings:mt-8 prose-headings:font-semibold prose-headings:text-black prose-h1:text-5xl prose-h2:text-4xl prose-h3:text-3xl prose-h4:te
      {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 className="prose prose-headings:mt-8 prose-headings:font-semibold prose-headings:text-black prose-h1:text-5xl prose-h2:text-4xl prose-h3:text-3xl prose-h4:te
      {children}
    </div>
  )
}
```



```
export default function MdxLayout({ children }) {
  // Create any shared layout or styles here
  return (
```

```
    <div className="prose prose-headings:mt-8 prose-headings:font-semibold prose-headings:text-black prose-h1:text-5xl prose-h2:text-4xl prose-h3:text-3xl prose-h4:te
      {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>

}
```

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

`@next/mdx` **does** allow you to use exports like any other JavaScript component:

```
export const metadata = {
  author: 'John Doe',
}

# Blog post
```

Metadata can now be referenced outside of the MDX file:

```
import BlogPost, { metadata } from '@/content/blog-post.mdx'

export default function Page() {
  console.log('metadata: ', metadata)
  //=> { author: 'John Doe' }
  return <BlogPost />
}
```


```
import BlogPost, { metadata } from '@/content/blog-post.mdx'

export default function Page() {
  console.log('metadata: ', metadata)
  //=> { author: 'John Doe' }
  return <BlogPost />
}
```

```
import BlogPost, { metadata } from '@/content/blog-post.mdx'

export default function Page() {
  console.log('metadata: ', metadata)
  //=> { author: 'John Doe' }
  return <BlogPost />
}
```

```
import BlogPost, { metadata } from '@/content/blog-post.mdx'

export default function Page() {
  console.log('metadata: ', metadata)
  //=> { author: 'John Doe' }
  return <BlogPost />
}
```


A common use case for this is when you want to iterate over a collection of MDX and extract data. For example, creating a blog index page from all blog posts. You can use packages like Node's `fs` module or globby to read a directory of posts and extract the metadata.

> **Good to know**:
>
> - Using `fs`, `globby`, etc. can only be used server-side.
> - View the Portfolio Starter Kit template for a complete working example.

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', 'md', 'mdx', 'ts', 'tsx'],
  // Optionally, add any other Next.js config below
}

const withMDX = createMDX({
  // Add markdown plugins here, as desired
  options: {
    remarkPlugins: [remarkGfm],
    rehypePlugins: [],
  },
})

// Wrap MDX and Next.js config with each other
export default withMDX(nextConfig)
```

Remote MDX

If your 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 package 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 `/mdx-page-remote` route should display your rendered MDX.

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,
  },
})
```

`mdxRs` also accepts an object to configure how to transform mdx files.

```
module.exports = withMDX({
  experimental: {
    mdxRs: {
      jsxRuntime?: string            // Custom jsx runtime
      jsxImportSource?: string       // Custom jsx import source,
      mdxType?: 'gfm' | 'commonmark' // Configure what kind of mdx syntax will be used to parse & transform
    },
  },
})
```

Good to know:

This option is required when processing markdown and MDX while using Turbopack (`next dev --turbopack`).

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.
- If you are using TypeScript paths for imports such as `@/*`, you should update the `paths` object in `tsconfig.json` to include `src/`.

---

title: Custom Server description: Start a Next.js app programmatically using a custom 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 includes its own server with `next start` by default. If you have an existing backend, you can still use it with Next.js (this is not a custom server). A custom Next.js server allows you to programmatically start a server for custom patterns. The majority of the time, you will not need this approach. However, it's available if you need to eject.

> **Good to know**:
>
> - Before deciding to use a custom server, keep in mind that it should only be used when the integrated router of Next.js can't meet your app requirements. A custom server will remove important performance optimizations, like **Automatic Static Optimization**.
> - A custom server **cannot** be deployed on Vercel.
> - When using standalone output mode, it does not trace custom server files. This mode outputs a separate minimal `server.js` file, instead. These cannot be used together.

Take a look at the following example of a custom server:

```
import { createServer } from 'http'
import { parse } from 'url'
import next from 'next'

const port = parseInt(process.env.PORT || '3000', 10)
const dev = process.env.NODE_ENV !== 'production'
const app = next({ dev })
const handle = app.getRequestHandler()

app.prepare().then(() => {
  createServer((req, res) => {
    const parsedUrl = parse(req.url!, true)
    handle(req, res, parsedUrl)
  }).listen(port)

  console.log(
    `> Server listening at http://localhost:${port} as ${
      dev ? 'development' : process.env.NODE_ENV
    }`
  )
})
```

```
import { createServer } from 'http'
import { parse } from 'url'
import next from 'next'

const port = parseInt(process.env.PORT || '3000', 10)
const dev = process.env.NODE_ENV !== 'production'
const app = next({ dev })
const handle = app.getRequestHandler()

app.prepare().then(() => {
  createServer((req, res) => {
    const parsedUrl = parse(req.url, true)
    handle(req, res, parsedUrl)
  }).listen(port)

  console.log(
    `> Server listening at http://localhost:${port} as ${
      dev ? 'development' : process.env.NODE_ENV
    }`
  )
})
```

> `server.js` does not run through the Next.js Compiler or bundling process. Make sure the syntax and source code this file requires are compatible with the current Node.js version you are using. View an example.

To run the custom server, you'll need to update the `scripts` in `package.json` like so:

```
{
  "scripts": {
    "dev": "node server.js",
    "build": "next build",
    "start": "NODE_ENV=production node server.js"
  }
}
```

Alternatively, you can set up `nodemon` (example). The custom server uses the following import to connect the server with the Next.js application:

```
import next from 'next'

const app = next({})
```

The above `next` import is a function that receives an object with the following options:

Option	Type	Description
`conf`	`Object`	The same object you would use in `next.config.js`. Defaults to `{}`
`customServer`	`Boolean`	(*Optional*) Set to false when the server was created by Next.js
`dev`	`Boolean`	(*Optional*) Whether or not to launch Next.js in dev mode. Defaults to `false`
`dir`	`String`	(*Optional*) Location of the Next.js project. Defaults to `'.'`
`quiet`	`Boolean`	(*Optional*) Hide error messages containing server information. Defaults to `false`
`hostname`	`String`	(*Optional*) The hostname the server is running behind
`port`	`Number`	(*Optional*) The port the server is running behind
`httpServer`	`node:http#Server`	(*Optional*) The HTTP Server that Next.js is running behind
`turbo`	`Boolean`	(*Optional*) Enable Turbopack

The returned `app` can then be used to let Next.js handle requests as required.

Disabling file-system routing

By default, `Next` will serve each file in the `pages` folder under a pathname matching the filename. If your project uses a custom server, this behavior may result in the same content being served from multiple paths, which can present problems with SEO and UX.

To disable this behavior and prevent routing based on files in `pages`, open `next.config.js` and disable the `useFileSystemPublicRoutes` config:

```
module.exports = {
  useFileSystemPublicRoutes: false,
}
```

Note that `useFileSystemPublicRoutes` disables filename routes from SSR; client-side routing may still access those paths. When using this option, you should guard against navigation to routes you do not want programmatically.

You may also wish to configure the client-side router to disallow client-side redirects to filename routes; for that refer to `router.beforePopState`.

---

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. related: title: Next Steps description: See the API reference for more information on how to use Draft Mode. links: - app/api-reference/functions/draft-mode

Draft Mode allows you to preview draft content from your headless CMS in your Next.js application. This is useful for static pages that are generated at build time as it allows you to switch to dynamic rendering and see the draft changes without having to rebuild your entire site.

This page walks through how to enable and use Draft Mode.

Step 1: Create a Route Handler

Create a Route Handler. It can have any name, for example, `app/api/draft/route.ts`.

```
export async function GET(request: Request) {
  return new Response('')
}
```

```
export async function GET() {
  return new Response('')
}
```

Then, import the `draftMode` function and call the `enable()` method.

```
import { draftMode } from 'next/headers'

export async function GET(request: Request) {
  const draft = await draftMode()
  draft.enable()
  return new Response('Draft mode is enabled')
}
```

```
import { draftMode } from 'next/headers'

export async function GET(request) {
  const draft = await draftMode()
  draft.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 and change the behavior of statically generated pages.

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`.

Step 2: Access the Route Handler from your Headless CMS

> 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. The specific steps will vary depending on which headless CMS you're using.

To securely access the Route Handler from your headless CMS:

1. 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.
2. If your headless CMS supports setting custom draft URLs, specify a draft URL (this assumes that your Route Handler is located at `app/api/draft/route.ts`). For example:

```
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/one`, then you should use `&slug=/posts/one`.

   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}`

3. In your 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`:

```
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
  const draft = await draftMode()
  draft.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)
}
```

```
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
  const draft = await draftMode()
  draft.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 3: Preview the Draft Content

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 } = await 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 } = await 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>
  )
}
```

If you access the draft Route Handler (with `secret` and `slug`) from your headless CMS or manually using the URL, 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.

---

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';
    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';
    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 async function Page() {
  const nonce = (await 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 async function Page() {
  const nonce = (await 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';
    upgrade-insecure-requests;
`

module.exports = {
  async headers() {
    return [
      {
        source: '/(.*)',
        headers: [
          {
            key: 'Content-Security-Policy',
            value: cspHeader.replace(/\n/g, ''),
          },
        ],
```

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

Version History

We recommend using `v13.4.20+` of Next.js to properly handle and apply nonces.

---

title: Debugging description: Learn how to debug your Next.js application with VS Code, Chrome DevTools, or Firefox DevTools.

{/* 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 documentation explains how you can debug your Next.js frontend and backend code with full source maps support using the VS Code debugger, Chrome DevTools, or Firefox DevTools.

Any debugger that can attach to Node.js can also be used to debug a Next.js application. You can find more details in the Node.js Debugging Guide.

Debugging with VS Code

Create a file named `.vscode/launch.json` at the root of your project with the following content:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Next.js: debug server-side",
      "type": "node-terminal",
      "request": "launch",
      "command": "npm run dev"
    },
    {
      "name": "Next.js: debug client-side",
      "type": "chrome",
      "request": "launch",
      "url": "http://localhost:3000"
    },
    {
      "name": "Next.js: debug client-side (Firefox)",
      "type": "firefox",
      "request": "launch",
      "url": "http://localhost:3000",
      "reAttach": true,
      "pathMappings": [
        {
          "url": "webpack://_N_E",
          "path": "${workspaceFolder}"
        }
      ]
    },
    {
      "name": "Next.js: debug full stack",
      "type": "node",
      "request": "launch",
      "program": "${workspaceFolder}/node_modules/.bin/next",
      "runtimeArgs": ["--inspect"],
      "skipFiles": ["<node_internals>/**"],
      "serverReadyAction": {
        "action": "debugWithEdge",
        "killOnServerStop": true,
        "pattern": "- Local:.+(https?://.+)",
        "uriFormat": "%s",
        "webRoot": "${workspaceFolder}"
      }
    }
  ]
}
```

> **Note**: To use Firefox debugging in VS Code, you'll need to install the Firefox Debugger extension.

`npm run dev` can be replaced with `yarn dev` if you're using Yarn or `pnpm dev` if you're using pnpm.

If you're changing the port number your application starts on, replace the `3000` in `http://localhost:3000` with the port you're using instead.

If you're running Next.js from a directory other than root (for example, if you're using Turborepo) then you need to add `cwd` to the server-side and full stack debugging tasks. For example, `"cwd": "${workspaceFolder}/apps/web"`.

Now go to the Debug panel (`Ctrl+Shift+D` on Windows/Linux, ⇧+⌘+D on macOS), select a launch configuration, then press `F5` or select **Debug: Start Debugging** from the Command Palette to start your debugging session.

Using the Debugger in Jetbrains WebStorm

Click the drop down menu listing the runtime configuration, and click `Edit Configurations...`. Create a `JavaScript Debug` debug configuration with `http://localhost:3000` as the URL. Customize to your liking (e.g. Browser for debugging, store as project file), and click `OK`. Run this debug configuration, and the selected browser should automatically open. At this point, you should have 2 applications in debug mode: the NextJS node application, and the client/browser application.

Debugging with Browser DevTools

Client-side code

Start your development server as usual by running `next dev`, `npm run dev`, or `yarn dev`. Once the server starts, open `http://localhost:3000` (or your alternate URL) in your preferred browser.

For Chrome:

- Open Chrome's Developer Tools (`Ctrl+Shift+J` on Windows/Linux, `⌥+⌘+I` on macOS)
- Go to the **Sources** tab

For Firefox:

- Open Firefox's Developer Tools (`Ctrl+Shift+I` on Windows/Linux, `⌥+⌘+I` on macOS)
- Go to the **Debugger** tab

In either browser, any time your client-side code reaches a `debugger` statement, code execution will pause and that file will appear in the debug area. You can also search for files to set breakpoints manually:

- In Chrome: Press `Ctrl+P` on Windows/Linux or `⌘+P` on macOS
- In Firefox: Press `Ctrl+P` on Windows/Linux or `⌘+P` on macOS, or use the file tree in the left panel

Note that when searching, your source files will have paths starting with `webpack://_N_E/./`.

Server-side code

To debug server-side Next.js code with browser DevTools, you need to pass the `--inspect` flag to the underlying Node.js process:

```
NODE_OPTIONS='--inspect' next dev
```

> **Good to know**: Use `NODE_OPTIONS='--inspect=0.0.0.0'` to allow remote debugging access outside localhost, such as when running the app in a Docker container.

If you're using `npm run dev` or `yarn dev` then you should update the `dev` script on your `package.json`:

```
{
  "scripts": {
    "dev": "NODE_OPTIONS='--inspect' next dev"
  }
}
```

Launching the Next.js dev server with the `--inspect` flag will look something like this:

```
Debugger listening on ws://127.0.0.1:9229/0cf90313-350d-4466-a748-cd60f4e47c95
For help, see: https://nodejs.org/en/docs/inspector
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
```

For Chrome:

1. Open a new tab and visit `chrome://inspect`
2. Look for your Next.js application in the **Remote Target** section
3. Click **inspect** to open a separate DevTools window
4. Go to the **Sources** tab

For Firefox:

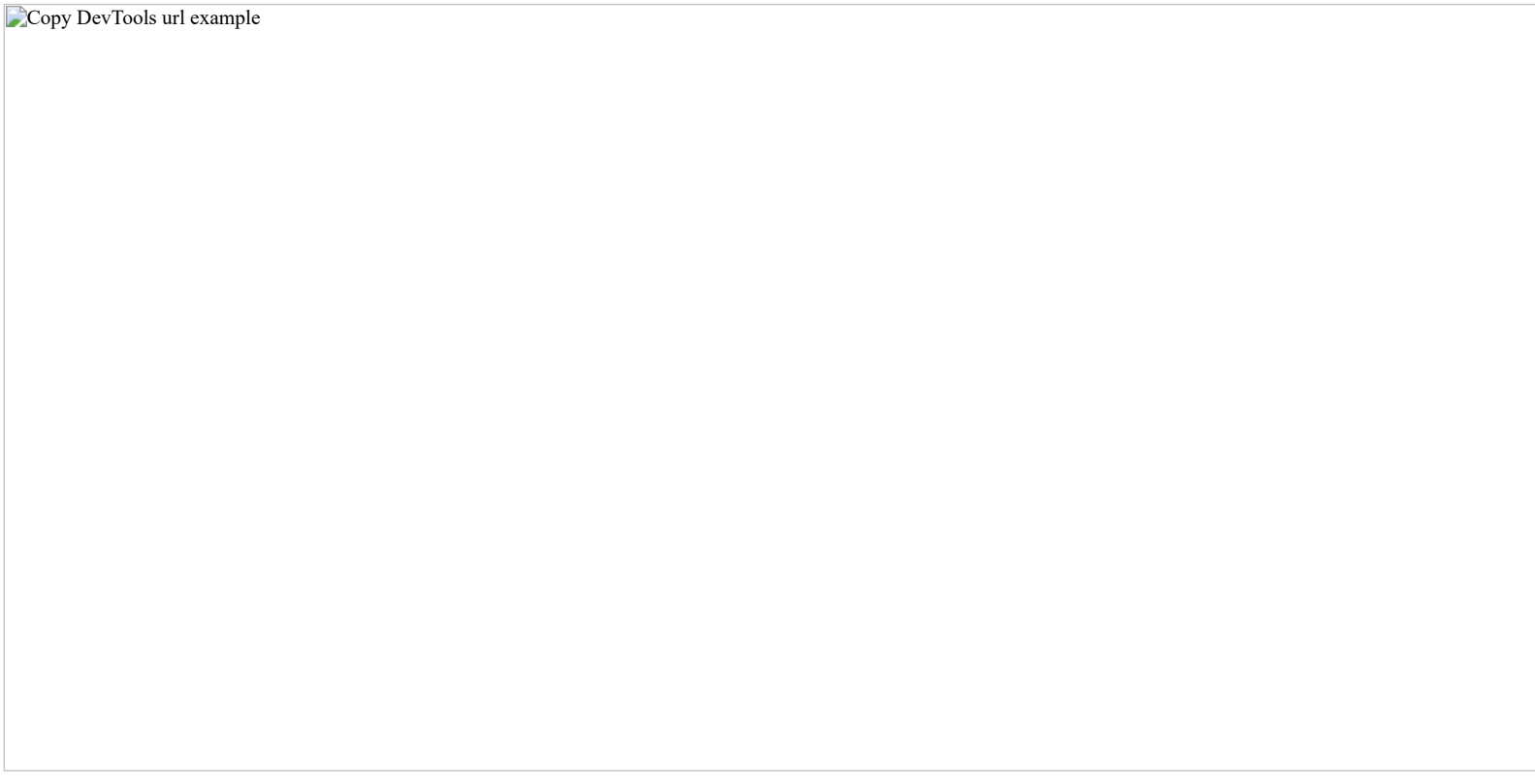
1. Open a new tab and visit `about:debugging`
2. Click **This Firefox** in the left sidebar
3. Under **Remote Targets**, find your Next.js application
4. Click **Inspect** to open the debugger
5. Go to the **Debugger** tab

Debugging server-side code works similarly to client-side debugging. When searching for files (`Ctrl+P`/`⌘+P`), your source files will have paths starting with `webpack://{application-name}/./` (where `{application-name}` will be replaced with the name of your application according to your `package.json` file).

Inspect Server Errors with Browser DevTools

When you encounter an error, inspecting the source code can help trace the root cause of errors.

Next.js will display a Node.js logo like a green button on the dev overlay. By clicking that button, the DevTools URL is copied to your clipboard. You can open a new browser tab with that URL to inspect the Next.js server process.



Debugging on Windows

Windows users may run into an issue when using `NODE_OPTIONS='--inspect'` as that syntax is not supported on Windows platforms. To get around this, install the `cross-env` package as a development dependency (`-D` with `npm` and `yarn`) and replace the `dev` script with the following.

```
{
  "scripts": {
    "dev": "cross-env NODE_OPTIONS='--inspect' next dev"
  }
}
```

`cross-env` will set the `NODE_OPTIONS` environment variable regardless of which platform you are on (including Mac, Linux, and Windows) and allow you to debug consistently across devices and operating systems.

> **Good to know**: Ensure Windows Defender is disabled on your machine. This external service will check *every file read*, which has been reported to greatly increase Fast Refresh time with `next dev`. This is a known issue, not related to Next.js, but it does affect Next.js development.

More information

To learn more about how to use a JavaScript debugger, take a look at the following documentation:

- Node.js debugging in VS Code: Breakpoints
- Chrome DevTools: Debug JavaScript
- Firefox DevTools: Debugger

---

title: Progressive Web Applications (PWA) description: Learn how to build a Progressive Web Application (PWA) with Next.js. related: links: - app/api-reference/file-conventions/metadata/manifest

Progressive Web Applications (PWAs) offer the reach and accessibility of web applications combined with the features and user experience of native mobile apps. With Next.js, you can create PWAs that provide a seamless, app-like experience across all platforms without the need for multiple codebases or app store approvals.

PWAs allow you to:

- Deploy updates instantly without waiting for app store approval
- Create cross-platform applications with a single codebase
- Provide native-like features such as home screen installation and push notifications

Creating a PWA with Next.js

1. Creating the Web App Manifest

Next.js provides built-in support for creating a web app manifest using the App Router. You can create either a static or dynamic manifest file:

For example, create a `app/manifest.ts` or `app/manifest.json` file:

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

export default function manifest(): MetadataRoute.Manifest {
  return {
    name: 'Next.js PWA',
    short_name: 'NextPWA',
    description: 'A Progressive Web App built with Next.js',
    start_url: '/',
    display: 'standalone',
    background_color: '#ffffff',
    theme_color: '#000000',
    icons: [
      {
        src: '/icon-192x192.png',
        sizes: '192x192',
        type: 'image/png',
      },
      {
        src: '/icon-512x512.png',
        sizes: '512x512',
        type: 'image/png',
      },
    ],
  }
}
```

```
export default function manifest() {
  return {
    name: 'Next.js PWA',
    short_name: 'NextPWA',
    description: 'A Progressive Web App built with Next.js',
    start_url: '/',
    display: 'standalone',
    background_color: '#ffffff',
    theme_color: '#000000',
    icons: [
      {
        src: '/icon-192x192.png',
        sizes: '192x192',
        type: 'image/png',
      },
      {
        src: '/icon-512x512.png',
        sizes: '512x512',
        type: 'image/png',
      },
    ],
  }
}
```

This file should contain information about the name, icons, and how it should be displayed as an icon on the user's device. This will allow users to install your PWA on their home screen, providing a native app-like experience.

You can use tools like favicon generators to create the different icon sets and place the generated files in your `public/` folder.

2. Implementing Web Push Notifications

Web Push Notifications are supported with all modern browsers, including:

- iOS 16.4+ for applications installed to the home screen
- Safari 16 for macOS 13 or later
- Chromium based browsers
- Firefox

This makes PWAs a viable alternative to native apps. Notably, you can trigger install prompts without needing offline support.

Web Push Notifications allow you to re-engage users even when they're not actively using your app. Here's how to implement them in a Next.js application:

First, let's create the main page component in `app/page.tsx`. We'll break it down into smaller parts for better understanding. First, we'll add some of the imports and utilities we'll need. It's okay that the referenced Server Actions do not yet exist:

```
'use client'

import { useState, useEffect } from 'react'
import { subscribeUser, unsubscribeUser, sendNotification } from './actions'

function urlBase64ToUint8Array(base64String: string) {
  const padding = '='.repeat((4 - (base64String.length % 4)) % 4)
  const base64 = (base64String + padding)
    .replace(/\\-/g, '+')
    .replace(/_/g, '/')

  const rawData = window.atob(base64)
  const outputArray = new Uint8Array(rawData.length)

  for (let i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i)
  }
  return outputArray
}
```

```
'use client'

import { useState, useEffect } from 'react'
import { subscribeUser, unsubscribeUser, sendNotification } from './actions'

function urlBase64ToUint8Array(base64String) {
  const padding = '='.repeat((4 - (base64String.length % 4)) % 4)
  const base64 = (base64String + padding)
    .replace(/\\-/g, '+')
    .replace(/_/g, '/')

  const rawData = window.atob(base64)
  const outputArray = new Uint8Array(rawData.length)

  for (let i = 0; i < rawData.length; ++i) {
    outputArray[i] = rawData.charCodeAt(i)
  }
  return outputArray
}
```

Let's now add a component to manage subscribing, unsubscribing, and sending push notifications.

```
function PushNotificationManager() {
  const [isSupported, setIsSupported] = useState(false)
  const [subscription, setSubscription] = useState<PushSubscription | null>(
    null
  )
  const [message, setMessage] = useState('')

  useEffect(() => {
    if ('serviceWorker' in navigator && 'PushManager' in window) {
      setIsSupported(true)
      registerServiceWorker()
    }
  }, [])

  async function registerServiceWorker() {
    const registration = await navigator.serviceWorker.register('/sw.js', {
      scope: '/',
      updateViaCache: 'none',
    })
    const sub = await registration.pushManager.getSubscription()
    setSubscription(sub)
  }

  async function subscribeToPush() {
    const registration = await navigator.serviceWorker.ready
    const sub = await registration.pushManager.subscribe({
      userVisibleOnly: true,
      applicationServerKey: urlBase64ToUint8Array(
        process.env.NEXT_PUBLIC_VAPID_PUBLIC_KEY!
      ),
    })
    setSubscription(sub)
    await subscribeUser(sub)
  }

  async function unsubscribeFromPush() {
    await subscription?.unsubscribe()
    setSubscription(null)
    await unsubscribeUser()
  }

  async function sendTestNotification() {
    if (subscription) {
      await sendNotification(message)
      setMessage('')
    }
  }

  if (!isSupported) {
    return <p>Push notifications are not supported in this browser.</p>
  }
```

```
  return (
    <div>
      <h3>Push Notifications</h3>
      {subscription ? (
        <>
          <p>You are subscribed to push notifications.</p>
          <button onClick={unsubscribeFromPush}>Unsubscribe</button>
          <input
            type="text"
            placeholder="Enter notification message"
            value={message}
            onChange={(e) => setMessage(e.target.value)}
          />
          <button onClick={sendTestNotification}>Send Test</button>
        </>
      ) : (
        <>
          <p>You are not subscribed to push notifications.</p>
          <button onClick={subscribeToPush}>Subscribe</button>
        </>
      )}
    </div>
  )
}

function PushNotificationManager() {
  const [isSupported, setIsSupported] = useState(false);
  const [subscription, setSubscription] = useState(null);
  const [message, setMessage] = useState('');

  useEffect(() => {
    if ('serviceWorker' in navigator && 'PushManager' in window) {
      setIsSupported(true);
      registerServiceWorker();
    }
  }, []);

  async function registerServiceWorker() {
    const registration = await navigator.serviceWorker.register('/sw.js', {
      scope: '/',
      updateViaCache: 'none',
    });
    const sub = await registration.pushManager.getSubscription();
    setSubscription(sub);
  }

  async function subscribeToPush() {
    const registration = await navigator.serviceWorker.ready;
    const sub = await registration.pushManager.subscribe({
      userVisibleOnly: true,
      applicationServerKey: urlBase64ToUint8Array(
        process.env.NEXT_PUBLIC_VAPID_PUBLIC_KEY!
      ),
    });
    setSubscription(sub);
    await subscribeUser(sub);
  }

  async function unsubscribeFromPush() {
    await subscription?.unsubscribe();
    setSubscription(null);
    await unsubscribeUser();
  }

  async function sendTestNotification() {
    if (subscription) {
      await sendNotification(message);
      setMessage('');
    }
  }

  if (!isSupported) {
    return <p>Push notifications are not supported in this browser.</p>;
  }

  return (
    <div>
      <h3>Push Notifications</h3>
      {subscription ? (
        <>
          <p>You are subscribed to push notifications.</p>
          <button onClick={unsubscribeFromPush}>Unsubscribe</button>
          <input
            type="text"
            placeholder="Enter notification message"
            value={message}
            onChange={(e) => setMessage(e.target.value)}
          />
          <button onClick={sendTestNotification}>Send Test</button>
        </>
      ) : (
        <>
          <p>You are not subscribed to push notifications.</p>
          <button onClick={subscribeToPush}>Subscribe</button>
        </>
      )}
    </div>
  );
}
```

Finally, let’s create a component to show a message for iOS devices to instruct them to install to their home screen, and only show this if the app is not already installed.

```
function InstallPrompt() {
  const [isIOS, setIsIOS] = useState(false)
  const [isStandalone, setIsStandalone] = useState(false)

  useEffect(() => {
    setIsIOS(
      /iPad|iPhone|iPod/.test(navigator.userAgent) && !(window as any).MSStream
    )
```

```
    setIsStandalone(window.matchMedia('(display-mode: standalone)').matches)
  }, [])

  if (isStandalone) {
    return null // Don't show install button if already installed
  }

  return (
    <div>
      <h3>Install App</h3>
      <button>Add to Home Screen</button>
      {isIOS && (
        <p>
          To install this app on your iOS device, tap the share button
          <span role="img" aria-label="share icon">
            {' '}
            ⎋{' '}
          </span>
          and then "Add to Home Screen"
          <span role="img" aria-label="plus icon">
            {' '}
            ➕{' '}
          </span>.
        </p>
      )}
    </div>
  )
}

export default function Page() {
  return (
    <div>
      <PushNotificationManager />
      <InstallPrompt />
    </div>
  )
}

function InstallPrompt() {
  const [isIOS, setIsIOS] = useState(false);
  const [isStandalone, setIsStandalone] = useState(false);

  useEffect(() => {
    setIsIOS(
      /iPad|iPhone|iPod/.test(navigator.userAgent) && !(window as any).MSStream
    );

    setIsStandalone(window.matchMedia('(display-mode: standalone)').matches);
  }, []);

  if (isStandalone) {
    return null; // Don't show install button if already installed
  }

  return (
    <div>
      <h3>Install App</h3>
      <button>Add to Home Screen</button>
      {isIOS && (
        <p>
          To install this app on your iOS device, tap the share button
          <span role="img" aria-label="share icon">
            {' '}
            ⎋{' '}
          </span>
          and then "Add to Home Screen"
          <span role="img" aria-label="plus icon">
            {' '}
            ➕{' '}
          </span>
          .
        </p>
      )}
    </div>
  );
}

export default function Page() {
  return (
    <div>
      <PushNotificationManager />
      <InstallPrompt />
    </div>
  );
}
```

Now, let's create the Server Actions which this file calls.

3. Implementing Server Actions

Create a new file to contain your actions at `app/actions.ts`. This file will handle creating subscriptions, deleting subscriptions, and sending notifications.

```
'use server'

import webpush from 'web-push'

webpush.setVapidDetails(
  '<mailto:your-email@example.com>',
  process.env.NEXT_PUBLIC_VAPID_PUBLIC_KEY!,
  process.env.VAPID_PRIVATE_KEY!
)

let subscription: PushSubscription | null = null

export async function subscribeUser(sub: PushSubscription) {
  subscription = sub
  // In a production environment, you would want to store the subscription in a database
  // For example: await db.subscriptions.create({ data: sub })
  return { success: true }
}
```

```
export async function unsubscribeUser() {
  subscription = null
  // In a production environment, you would want to remove the subscription from the database
  // For example: await db.subscriptions.delete({ where: { ... } })
  return { success: true }
}

export async function sendNotification(message: string) {
  if (!subscription) {
    throw new Error('No subscription available')
  }

  try {
    await webpush.sendNotification(
      subscription,
      JSON.stringify({
        title: 'Test Notification',
        body: message,
        icon: '/icon.png',
      })
    )
    return { success: true }
  } catch (error) {
    console.error('Error sending push notification:', error)
    return { success: false, error: 'Failed to send notification' }
  }
}
```

```
'use server';

import webpush from 'web-push';

webpush.setVapidDetails(
  '<mailto:your-email@example.com>',
  process.env.NEXT_PUBLIC_VAPID_PUBLIC_KEY!,
  process.env.VAPID_PRIVATE_KEY!
);

let subscription= null;

export async function subscribeUser(sub) {
  subscription = sub;
  // In a production environment, you would want to store the subscription in a database
  // For example: await db.subscriptions.create({ data: sub })
  return { success: true };
}

export async function unsubscribeUser() {
  subscription = null;
  // In a production environment, you would want to remove the subscription from the database
  // For example: await db.subscriptions.delete({ where: { ... } })
  return { success: true };
}

export async function sendNotification(message) {
  if (!subscription) {
    throw new Error('No subscription available');
  }

  try {
    await webpush.sendNotification(
      subscription,
      JSON.stringify({
        title: 'Test Notification',
        body: message,
        icon: '/icon.png',
      })
    );
    return { success: true };
  } catch (error) {
    console.error('Error sending push notification:', error);
    return { success: false, error: 'Failed to send notification' };
  }
}
```

Sending a notification will be handled by our service worker, created in step 5.

In a production environment, you would want to store the subscription in a database for persistence across server restarts and to manage multiple users' subscriptions.

4. Generating VAPID Keys

To use the Web Push API, you need to generate VAPID keys.

Create a script file, e.g., `generate-vapid-keys.js`:

```
const webpush = require('web-push')
const vapidKeys = webpush.generateVAPIDKeys()

console.log('Paste the following keys in your .env file:')
console.log('--------------------')
console.log('NEXT_PUBLIC_VAPID_PUBLIC_KEY=', vapidKeys.publicKey)
console.log('VAPID_PRIVATE_KEY=', vapidKeys.privateKey)
```

Run this script with Node.js to generate your VAPID keys:

```
node generate-vapid-keys.js
```

Copy the output and paste it into your `.env` file.

5. Creating a Service Worker

Create a `public/sw.js` file for your service worker:

```
self.addEventListener('push', function (event) {
  if (event.data) {
    const data = event.data.json()
    const options = {
```

```
      body: data.body,
      icon: data.icon || '/icon.png',
      badge: '/badge.png',
      vibrate: [100, 50, 100],
      data: {
        dateOfArrival: Date.now(),
        primaryKey: '2',
      },
    }
    event.waitUntil(self.registration.showNotification(data.title, options))
  }
})

self.addEventListener('notificationclick', function (event) {
  console.log('Notification click received.')
  event.notification.close()
  event.waitUntil(clients.openWindow('<https://your-website.com>'))
})
```

This service worker supports custom images and notifications. It handles incoming push events and notification clicks.

- You can set custom icons for notifications using the `icon` and `badge` properties.
- The `vibrate` pattern can be adjusted to create custom vibration alerts on supported devices.
- Additional data can be attached to the notification using the `data` property.

Remember to test your service worker thoroughly to ensure it behaves as expected across different devices and browsers. Also, make sure to update the `'https://your-website.com'` link in the `notificationclick` event listener to the appropriate URL for your application.

6. Adding to Home Screen

The `InstallPrompt` component defined in step 2 shows a message for iOS devices to instruct them to install to their home screen.

To ensure your application can be installed to a mobile home screen, you must have:

1. A valid web app manifest (created in step 1)
2. The website served over HTTPS

Modern browsers will automatically show an installation prompt to users when these criteria are met. You can provide a custom installation button with `beforeinstallprompt`, however, we do not recommend this as it is not cross browser and platform (does not work on Safari iOS).

7. Testing Locally

To ensure you can view notifications locally, ensure that:

- You are running locally with HTTPS
  - Use `next dev --experimental-https` for testing
- Your browser (Chrome, Safari, Firefox) has notifications enabled
  - When prompted locally, accept permissions to use notifications
  - Ensure notifications are not disabled globally for the entire browser
  - If you are still not seeing notifications, try using another browser to debug

8. Securing your application

Security is a crucial aspect of any web application, especially for PWAs. Next.js allows you to configure security headers using the `next.config.js` file. For example:

```
module.exports = {
  async headers() {
    return [
      {
        source: '/(.*)',
        headers: [
          {
            key: 'X-Content-Type-Options',
            value: 'nosniff',
          },
          {
            key: 'X-Frame-Options',
            value: 'DENY',
          },
          {
            key: 'Referrer-Policy',
            value: 'strict-origin-when-cross-origin',
          },
        ],
      },
      {
        source: '/sw.js',
        headers: [
          {
            key: 'Content-Type',
            value: 'application/javascript; charset=utf-8',
          },
          {
            key: 'Cache-Control',
            value: 'no-cache, no-store, must-revalidate',
          },
          {
            key: 'Content-Security-Policy',
            value: "default-src 'self'; script-src 'self'",
          },
        ],
      },
    ]
  },
}
```

Let's go over each of these options:

1. Global Headers (applied to all routes):
    1. `X-Content-Type-Options: nosniff`: Prevents MIME type sniffing, reducing the risk of malicious file uploads.
    2. `X-Frame-Options: DENY`: Protects against clickjacking attacks by preventing your site from being embedded in iframes.
    3. `Referrer-Policy: strict-origin-when-cross-origin`: Controls how much referrer information is included with requests, balancing security and functionality.
2. Service Worker Specific Headers:

1. `Content-Type: application/javascript; charset=utf-8`: Ensures the service worker is interpreted correctly as JavaScript.
2. `Cache-Control: no-cache, no-store, must-revalidate`: Prevents caching of the service worker, ensuring users always get the latest version.
3. `Content-Security-Policy: default-src 'self'; script-src 'self'`: Implements a strict Content Security Policy for the service worker, only allowing scripts from the same origin.

Learn more about defining Content Security Policies with Next.js.

Next Steps

1. **Exploring PWA Capabilities**: PWAs can leverage various web APIs to provide advanced functionality. Consider exploring features like background sync, periodic background sync, or the File System Access API to enhance your application. For inspiration and up-to-date information on PWA capabilities, you can refer to resources like What PWA Can Do Today.
2. **Static Exports:** If your application requires not running a server, and instead using a static export of files, you can update the Next.js configuration to enable this change. Learn more in the Next.js Static Export documentation. However, you will need to move from Server Actions to calling an external API, as well as moving your defined headers to your proxy.
3. **Offline Support**: To provide offline functionality, one option is Serwist with Next.js. You can find an example of how to integrate Serwist with Next.js in their documentation. **Note:** this plugin currently requires webpack configuration.
4. **Security Considerations**: Ensure that your service worker is properly secured. This includes using HTTPS, validating the source of push messages, and implementing proper error handling.
5. **User Experience**: Consider implementing progressive enhancement techniques to ensure your app works well even when certain PWA features are not supported by the user's browser.

---

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 @testing-library/dom
# or
yarn add -D vitest @vitejs/plugin-react jsdom @testing-library/react @testing-library/dom
# or
pnpm install -D vitest @vitejs/plugin-react jsdom @testing-library/react @testing-library/dom
# or
bun add -D vitest @vitejs/plugin-react jsdom @testing-library/react @testing-library/dom
```

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 Configuration 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.

{/* 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. */}

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/dom @testing-library/jest-dom ts-node
# or
yarn add -D jest jest-environment-jsdom @testing-library/react @testing-library/dom @testing-library/jest-dom ts-node
# or
pnpm install -D jest jest-environment-jsdom @testing-library/react @testing-library/dom @testing-library/jest-dom ts-node
```

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:

```ts
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)
```

```js
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)$': `<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`, 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` 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 function Page() {
  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 versions below 13.6.3 do not support TypeScript version 5 with `moduleResolution:"bundler"`. However, this issue has been resolved in Cypress version 13.6.3 and later. cypress v13.6.3

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:

```tsx
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```jsx
import Link from 'next/link'

export default function Page() {
  return (
    <div>
      <h1>About</h1>
      <Link href="/">Home</Link>
    </div>
  )
}
```

```tsx
import Link from 'next/link'

export default function Home() {
  return (
    <div>
      <h1>Home</h1>
      <Link href="/about">About</Link>
    </div>
  )
}
```

```jsx
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 conjunction 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:

```tsx
import { defineConfig } from 'cypress'

export default defineConfig({
  component: {
    devServer: {
      framework: 'next',
      bundler: 'webpack',
    },
  },
})
```

```jsx
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 four 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: Authentication description: Learn how to implement authentication in your Next.js application.

Understanding authentication is crucial for protecting your application's data. This page will guide you through what React and Next.js features to use to implement auth.

Before starting, it helps to break down the process into three concepts:

1. **Authentication**: Verifies if the user is who they say they are. It requires the user to prove their identity with something they have, such as a username and password.
2. **Session Management**: Tracks the user's auth state across requests.
3. **Authorization**: Decides what routes and data the user can access.

This diagram shows the authentication flow using React and Next.js features:

The examples on this page walk through basic username and password auth for educational purposes. While you can implement a custom auth solution, for increased security and simplicity, we recommend using an authentication library. These offer built-in solutions for authentication, session management, and authorization, as well as additional features such as social logins, multi-factor authentication, and role-based access control. You can find a list in the Auth Libraries section.

Authentication

Sign-up and login functionality

You can use the `<form>` element with React's Server Actions and `useFormState` to capture user credentials, validate form fields, and call your Authentication Provider's API or database.

Since Server Actions always execute on the server, they provide a secure environment for handling authentication logic.

Here are the steps to implement signup/login functionality:

1. Capture user credentials

To capture user credentials, create a form that invokes a Server Action on submission. For example, a signup form that accepts the user's name, email, and password:

```
import { signup } from '@/app/actions/auth'

export function SignupForm() {
  return (
    <form action={signup}>
      <div>
        <label htmlFor="name">Name</label>
        <input id="name" name="name" placeholder="Name" />
      </div>
      <div>
        <label htmlFor="email">Email</label>
        <input id="email" name="email" type="email" placeholder="Email" />
      </div>
      <div>
        <label htmlFor="password">Password</label>
        <input id="password" name="password" type="password" />
      </div>
      <button type="submit">Sign Up</button>
    </form>
  )
}
```

```
import { signup } from '@/app/actions/auth'

export function SignupForm() {
  return (
    <form action={signup}>
      <div>
        <label htmlFor="name">Name</label>
        <input id="name" name="name" placeholder="Name" />
      </div>
      <div>
        <label htmlFor="email">Email</label>
        <input id="email" name="email" type="email" placeholder="Email" />
      </div>
      <div>
        <label htmlFor="password">Password</label>
        <input id="password" name="password" type="password" />
      </div>
      <button type="submit">Sign Up</button>
    </form>
  )
}

export async function signup(formData: FormData) {}
```

```
export async function signup(formData) {}
```

2. Validate form fields on the server

Use the Server Action to validate the form fields on the server. If your authentication provider doesn't provide form validation, you can use a schema validation library like Zod or Yup.

Using Zod as an example, you can define a form schema with appropriate error messages:

```
import { z } from 'zod'

export const SignupFormSchema = z.object({
  name: z
    .string()
    .min(2, { message: 'Name must be at least 2 characters long.' })
    .trim(),
  email: z.string().email({ message: 'Please enter a valid email.' }).trim(),
  password: z
    .string()
    .min(8, { message: 'Be at least 8 characters long' })
    .regex(/[a-zA-Z]/, { message: 'Contain at least one letter.' })
    .regex(/[0-9]/, { message: 'Contain at least one number.' })
    .regex(/[^a-zA-Z0-9]/, {
      message: 'Contain at least one special character.',
    })
    .trim(),
})

export type FormState =
  | {
      errors?: {
        name?: string[]
        email?: string[]
        password?: string[]
      }
      message?: string
    }
  | undefined
```

```
import { z } from 'zod'

export const SignupFormSchema = z.object({
  name: z
    .string()
    .min(2, { message: 'Name must be at least 2 characters long.' })
    .trim(),
  email: z.string().email({ message: 'Please enter a valid email.' }).trim(),
  password: z
    .string()
    .min(8, { message: 'Be at least 8 characters long' })
    .regex(/[a-zA-Z]/, { message: 'Contain at least one letter.' })
    .regex(/[0-9]/, { message: 'Contain at least one number.' })
    .regex(/[^a-zA-Z0-9]/, {
      message: 'Contain at least one special character.',
```

```
    })
    .trim(),
})
```

To prevent unnecessary calls to your authentication provider's API or database, you can `return` early in the Server Action if any form fields do not match the defined schema.

```
import { SignupFormSchema, FormState } from '@/app/lib/definitions'

export async function signup(state: FormState, formData: FormData) {
  // Validate form fields
  const validatedFields = SignupFormSchema.safeParse({
    name: formData.get('name'),
    email: formData.get('email'),
    password: formData.get('password'),
  })

  // If any form fields are invalid, return early
  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
    }
  }

  // Call the provider or db to create a user...
}
```

```
import { SignupFormSchema } from '@/app/lib/definitions'

export async function signup(state, formData) {
  // Validate form fields
  const validatedFields = SignupFormSchema.safeParse({
    name: formData.get('name'),
    email: formData.get('email'),
    password: formData.get('password'),
  })

  // If any form fields are invalid, return early
  if (!validatedFields.success) {
    return {
      errors: validatedFields.error.flatten().fieldErrors,
    }
  }

  // Call the provider or db to create a user...
}
```

Back in your `<SignupForm />`, you can use React's `useFormState` hook to display validation errors while the form is submitting:

```
'use client'

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

export function SignupForm() {
  const [state, action] = useFormState(signup, undefined)

  return (
    <form action={action}>
      <div>
        <label htmlFor="name">Name</label>
        <input id="name" name="name" placeholder="Name" />
      </div>
      {state?.errors?.name && <p>{state.errors.name}</p>}

      <div>
        <label htmlFor="email">Email</label>
        <input id="email" name="email" placeholder="Email" />
      </div>
      {state?.errors?.email && <p>{state.errors.email}</p>}

      <div>
        <label htmlFor="password">Password</label>
        <input id="password" name="password" type="password" />
      </div>
      {state?.errors?.password && (
        <div>
          <p>Password must:</p>
          <ul>
            {state.errors.password.map((error) => (
              <li key={error}>- {error}</li>
            ))}
          </ul>
        </div>
      )}
      <SubmitButton />
    </form>
  )
}

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

  return (
    <button disabled={pending} type="submit">
      Sign Up
    </button>
  )
}
```

```
'use client'

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

export function SignupForm() {
  const [state, action] = useFormState(signup, undefined)

  return (
    <form action={action}>
      <div>
        <label htmlFor="name">Name</label>
        <input id="name" name="name" placeholder="John Doe" />
```

```
      </div>
      {state?.errors?.name && <p>{state.errors.name}</p>}

      <div>
        <label htmlFor="email">Email</label>
        <input id="email" name="email" placeholder="john@example.com" />
      </div>
      {state?.errors?.email && <p>{state.errors.email}</p>}

      <div>
        <label htmlFor="password">Password</label>
        <input id="password" name="password" type="password" />
      </div>
      {state?.errors?.password && (
        <div>
          <p>Password must:</p>
          <ul>
            {state.errors.password.map((error) => (
              <li key={error}>- {error}</li>
            ))}
          </ul>
        </div>
      )}
      <SubmitButton />
    </form>
  )
}

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

  return (
    <button disabled={pending} type="submit">
      Sign Up
    </button>
  )
}
```

Good to know:

- These examples use React's `useFormState` hook, which is bundled with the Next.js App Router. If you are using React 19, use `useActionState` instead. See the React docs for more information.
- In React 19, `useFormStatus` includes additional keys on the returned object, like data, method, and action. If you are not using React 19, only the `pending` key is available.
- In React 19, `useActionState` also includes a `pending` key on the returned state.
- Before mutating data, you should always ensure a user is also authorized to perform the action. See Authentication and Authorization.

3. Create a user or check user credentials

After validating form fields, you can create a new user account or check if the user exists by calling your authentication provider's API or database.

Continuing from the previous example:

```
export async function signup(state: FormState, formData: FormData) {
  // 1. Validate form fields
  // ...

  // 2. Prepare data for insertion into database
  const { name, email, password } = validatedFields.data
  // e.g. Hash the user's password before storing it
  const hashedPassword = await bcrypt.hash(password, 10)

  // 3. Insert the user into the database or call an Auth Library's API
  const data = await db
    .insert(users)
    .values({
      name,
      email,
      password: hashedPassword,
    })
    .returning({ id: users.id })

  const user = data[0]

  if (!user) {
    return {
      message: 'An error occurred while creating your account.',
    }
  }

  // TODO:
  // 4. Create user session
  // 5. Redirect user
}
```

```
export async function signup(state, formData) {
  // 1. Validate form fields
  // ...

  // 2. Prepare data for insertion into database
  const { name, email, password } = validatedFields.data
  // e.g. Hash the user's password before storing it
  const hashedPassword = await bcrypt.hash(password, 10)

  // 3. Insert the user into the database or call an Library API
  const data = await db
    .insert(users)
    .values({
      name,
      email,
      password: hashedPassword,
    })
    .returning({ id: users.id })

  const user = data[0]

  if (!user) {
    return {
      message: 'An error occurred while creating your account.',
```

```
    }
  }

  // TODO:
  // 4. Create user session
  // 5. Redirect user
}
```

After successfully creating the user account or verifying the user credentials, you can create a session to manage the user's auth state. Depending on your session management strategy, the session can be stored in a cookie or database, or both. Continue to the Session Management section to learn more.

Tips:

- The example above is verbose since it breaks down the authentication steps for the purpose of education. This highlights that implementing your own secure solution can quickly become complex. Consider using an Auth Library to simplify the process.
- To improve the user experience, you may want to check for duplicate emails or usernames earlier in the registration flow. For example, as the user types in a username or the input field loses focus. This can help prevent unnecessary form submissions and provide immediate feedback to the user. You can debounce requests with libraries such as use-debounce to manage the frequency of these checks.

Here are the steps to implement a sign-up and/or login form:

1. The user submits their credentials through a form.
2. The form sends a request that is handled by an API route.
3. Upon successful verification, the process is completed, indicating the user's successful authentication.
4. If verification is unsuccessful, an error message is shown.

Consider a login form where users can input their credentials:

```
import { FormEvent } from 'react'
import { useRouter } from 'next/router'

export default function LoginPage() {
  const router = useRouter()

  async function handleSubmit(event: FormEvent<HTMLFormElement>) {
    event.preventDefault()

    const formData = new FormData(event.currentTarget)
    const email = formData.get('email')
    const password = formData.get('password')

    const response = await fetch('/api/auth/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email, password }),
    })

    if (response.ok) {
      router.push('/profile')
    } else {
      // Handle errors
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" name="email" placeholder="Email" required />
      <input type="password" name="password" placeholder="Password" required />
      <button type="submit">Login</button>
    </form>
  )
}
```

```
import { FormEvent } from 'react'
import { useRouter } from 'next/router'

export default function LoginPage() {
  const router = useRouter()

  async function handleSubmit(event) {
    event.preventDefault()

    const formData = new FormData(event.currentTarget)
    const email = formData.get('email')
    const password = formData.get('password')

    const response = await fetch('/api/auth/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email, password }),
    })

    if (response.ok) {
      router.push('/profile')
    } else {
      // Handle errors
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" name="email" placeholder="Email" required />
      <input type="password" name="password" placeholder="Password" required />
      <button type="submit">Login</button>
    </form>
  )
}
```

The form above has two input fields for capturing the user's email and password. On submission, it triggers a function that sends a POST request to an API route (`/api/auth/login`).

You can then call your Authentication Provider's API in the API route to handle authentication:

```
import type { NextApiRequest, NextApiResponse } from 'next'
import { signIn } from '@/auth'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
```

```
) {
  try {
    const { email, password } = req.body
    await signIn('credentials', { email, password })

    res.status(200).json({ success: true })
  } catch (error) {
    if (error.type === 'CredentialsSignin') {
      res.status(401).json({ error: 'Invalid credentials.' })
    } else {
      res.status(500).json({ error: 'Something went wrong.' })
    }
  }
}
```

```
import { signIn } from '@/auth'

export default async function handler(req, res) {
  try {
    const { email, password } = req.body
    await signIn('credentials', { email, password })

    res.status(200).json({ success: true })
  } catch (error) {
    if (error.type === 'CredentialsSignin') {
      res.status(401).json({ error: 'Invalid credentials.' })
    } else {
      res.status(500).json({ error: 'Something went wrong.' })
    }
  }
}
```

Session Management

Session management ensures that the user's authenticated state is preserved across requests. It involves creating, storing, refreshing, and deleting sessions or tokens.

There are two types of sessions:

1. **Stateless**: Session data (or a token) is stored in the browser's cookies. The cookie is sent with each request, allowing the session to be verified on the server. This method is simpler, but can be less secure if not implemented correctly.
2. **Database**: Session data is stored in a database, with the user's browser only receiving the encrypted session ID. This method is more secure, but can be complex and use more server resources.

> **Good to know:** While you can use either method, or both, we recommend using session management library such as iron-session or Jose.

Stateless Sessions

To create and manage stateless sessions, there are a few steps you need to follow:

1. Generate a secret key, which will be used to sign your session, and store it as an environment variable.
2. Write logic to encrypt/decrypt session data using a session management library.
3. Manage cookies using the Next.js `cookies` API.

In addition to the above, consider adding functionality to update (or refresh) the session when the user returns to the application, and delete the session when the user logs out.

> **Good to know:** Check if your auth library includes session management.

1. Generating a secret key

There are a few ways you can generate secret key to sign your session. For example, you may choose to use the `openssl` command in your terminal:

```
openssl rand -base64 32
```

This command generates a 32-character random string that you can use as your secret key and store in your environment variables file:

```
SESSION_SECRET=your_secret_key
```

You can then reference this key in your session management logic:

```
const secretKey = process.env.SESSION_SECRET
```

2. Encrypting and decrypting sessions

Next, you can use your preferred session management library to encrypt and decrypt sessions. Continuing from the previous example, we'll use Jose (compatible with the Edge Runtime) and React's `server-only` package to ensure that your session management logic is only executed on the server.

```
import 'server-only'
import { SignJWT, jwtVerify } from 'jose'
import { SessionPayload } from '@/app/lib/definitions'

const secretKey = process.env.SESSION_SECRET
const encodedKey = new TextEncoder().encode(secretKey)

export async function encrypt(payload: SessionPayload) {
  return new SignJWT(payload)
    .setProtectedHeader({ alg: 'HS256' })
    .setIssuedAt()
    .setExpirationTime('7d')
    .sign(encodedKey)
}

export async function decrypt(session: string | undefined = '') {
  try {
    const { payload } = await jwtVerify(session, encodedKey, {
      algorithms: ['HS256'],
    })
    return payload
  } catch (error) {
    console.log('Failed to verify session')
  }
}
```

```
import 'server-only'
import { SignJWT, jwtVerify } from 'jose'

const secretKey = process.env.SESSION_SECRET
const encodedKey = new TextEncoder().encode(secretKey)

export async function encrypt(payload) {
  return new SignJWT(payload)
    .setProtectedHeader({ alg: 'HS256' })
    .setIssuedAt()
    .setExpirationTime('7d')
    .sign(encodedKey)
}

export async function decrypt(session) {
  try {
    const { payload } = await jwtVerify(session, encodedKey, {
      algorithms: ['HS256'],
    })
    return payload
  } catch (error) {
    console.log('Failed to verify session')
  }
}
```

Tips:

- The payload should contain the **minimum**, unique user data that'll be used in subsequent requests, such as the user's ID, role, etc. It should not contain personally identifiable information like phone number, email address, credit card information, etc, or sensitive data like passwords.

3. Setting cookies (recommended options)

To store the session in a cookie, use the Next.js `cookies` API. The cookie should be set on the server, and include the recommended options:

- **HttpOnly**: Prevents client-side JavaScript from accessing the cookie.
- **Secure**: Use https to send the cookie.
- **SameSite**: Specify whether the cookie can be sent with cross-site requests.
- **Max-Age or Expires**: Delete the cookie after a certain period.
- **Path**: Define the URL path for the cookie.

Please refer to MDN for more information on each of these options.

```
import 'server-only'
import { cookies } from 'next/headers'

export async function createSession(userId: string) {
  const expiresAt = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)
  const session = await encrypt({ userId, expiresAt })(await cookies()).set(
    'session',
    session,
    {
      httpOnly: true,
      secure: true,
      expires: expiresAt,
      sameSite: 'lax',
      path: '/',
    }
  )
}
```

```
import 'server-only'
import { cookies } from 'next/headers'

export async function createSession(userId) {
  const expiresAt = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)
  const session = await encrypt({ userId, expiresAt })
  const cookieStore = await cookies()

  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expiresAt,
    sameSite: 'lax',
    path: '/',
  })
}
```

Back in your Server Action, you can invoke the `createSession()` function, and use the `redirect()` API to redirect the user to the appropriate page:

```
import { createSession } from '@/app/lib/session'

export async function signup(state: FormState, formData: FormData) {
  // Previous steps:
  // 1. Validate form fields
  // 2. Prepare data for insertion into database
  // 3. Insert the user into the database or call an Library API

  // Current steps:
  // 4. Create user session
  await createSession(user.id)
  // 5. Redirect user
  redirect('/profile')
}
```

```
import { createSession } from '@/app/lib/session'

export async function signup(state, formData) {
  // Previous steps:
  // 1. Validate form fields
  // 2. Prepare data for insertion into database
  // 3. Insert the user into the database or call an Library API

  // Current steps:
  // 4. Create user session
  await createSession(user.id)
  // 5. Redirect user
  redirect('/profile')
}
```

Tips:

- **Cookies should be set on the server** to prevent client-side tampering.
- 📹 Watch: Learn more about stateless sessions and authentication with Next.js → YouTube (11 minutes).

Updating (or refreshing) sessions

You can also extend the session's expiration time. This is useful for keeping the user logged in after they access the application again. For example:

```
import 'server-only'
import { cookies } from 'next/headers'
import { decrypt } from '@/app/lib/session'

export async function updateSession() {
  const session = (await cookies()).get('session')?.value
  const payload = await decrypt(session)

  if (!session || !payload) {
    return null
  }

  const expires = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)

  const cookieStore = await cookies()
  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expires,
    sameSite: 'lax',
    path: '/',
  })
}
```

```
import 'server-only'
import { cookies } from 'next/headers'
import { decrypt } from '@/app/lib/session'

export async function updateSession() {
  const session = (await cookies()).get('session')?.value
  const payload = await decrypt(session)

  if (!session || !payload) {
    return null
  }

  const expires = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)(
    await cookies()
  ).set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expires,
    sameSite: 'lax',
    path: '/',
  })
}
```

> **Tip:** Check if your auth library supports refresh tokens, which can be used to extend the user's session.

Deleting the session

To delete the session, you can delete the cookie:

```
import 'server-only'
import { cookies } from 'next/headers'

export async function deleteSession() {
  const cookieStore = await cookies()
  cookieStore.delete('session')
}
```

```
import 'server-only'
import { cookies } from 'next/headers'

export async function deleteSession() {
  const cookieStore = await cookies()
  cookieStore.delete('session')
}
```

Then you can reuse the `deleteSession()` function in your application, for example, on logout:

```
import { cookies } from 'next/headers'
import { deleteSession } from '@/app/lib/session'

export async function logout() {
  deleteSession()
  redirect('/login')
}
```

```
import { cookies } from 'next/headers'
import { deleteSession } from '@/app/lib/session'

export async function logout() {
  deleteSession()
  redirect('/login')
}
```

Setting and deleting cookies

You can use API Routes to set the session as a cookie on the server:

```
import { serialize } from 'cookie'
import type { NextApiRequest, NextApiResponse } from 'next'
import { encrypt } from '@/app/lib/session'

export default function handler(req: NextApiRequest, res: NextApiResponse) {
  const sessionData = req.body
  const encryptedSessionData = encrypt(sessionData)
```

```
  const cookie = serialize('session', encryptedSessionData, {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production',
    maxAge: 60 * 60 * 24 * 7, // One week
    path: '/',
  })
  res.setHeader('Set-Cookie', cookie)
  res.status(200).json({ message: 'Successfully set cookie!' })
}
```

```
import { serialize } from 'cookie'
import { encrypt } from '@/app/lib/session'

export default function handler(req, res) {
  const sessionData = req.body
  const encryptedSessionData = encrypt(sessionData)

  const cookie = serialize('session', encryptedSessionData, {
    httpOnly: true,
    secure: process.env.NODE_ENV === 'production',
    maxAge: 60 * 60 * 24 * 7, // One week
    path: '/',
  })
  res.setHeader('Set-Cookie', cookie)
  res.status(200).json({ message: 'Successfully set cookie!' })
}
```

Database Sessions

To create and manage database sessions, you'll need to follow these steps:

1. Create a table in your database to store session and data (or check if your Auth Library handles this).
2. Implement functionality to insert, update, and delete sessions
3. Encrypt the session ID before storing it in the user's browser, and ensure the database and cookie stay in sync (this is optional, but recommended for optimistic auth checks in Middleware).

For example:

```
import cookies from 'next/headers'
import { db } from '@/app/lib/db'
import { encrypt } from '@/app/lib/session'

export async function createSession(id: number) {
  const expiresAt = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)

  // 1. Create a session in the database
  const data = await db
    .insert(sessions)
    .values({
      userId: id,
      expiresAt,
    })
    // Return the session ID
    .returning({ id: sessions.id })

  const sessionId = data[0].id

  // 2. Encrypt the session ID
  const session = await encrypt({ sessionId, expiresAt })

  // 3. Store the session in cookies for optimistic auth checks
  const cookieStore = await cookies()
  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expiresAt,
    sameSite: 'lax',
    path: '/',
  })
}
```

```
import cookies from 'next/headers'
import { db } from '@/app/lib/db'
import { encrypt } from '@/app/lib/session'

export async function createSession(id) {
  const expiresAt = new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)

  // 1. Create a session in the database
  const data = await db
    .insert(sessions)
    .values({
      userId: id,
      expiresAt,
    })
    // Return the session ID
    .returning({ id: sessions.id })

  const sessionId = data[0].id

  // 2. Encrypt the session ID
  const session = await encrypt({ sessionId, expiresAt })

  // 3. Store the session in cookies for optimistic auth checks
  const cookieStore = await cookies()
  cookieStore.set('session', session, {
    httpOnly: true,
    secure: true,
    expires: expiresAt,
    sameSite: 'lax',
    path: '/',
  })
}
```

Tips:

- For faster data retrieval, consider using a database like Vercel Redis. However, you can also keep the session data in your primary database, and combine data requests to reduce the number of queries.

- You may opt to use database sessions for more advanced use cases, such as keeping track of the last time a user logged in, or number of active devices, or give users the ability to log out of all devices.

After implementing session management, you'll need to add authorization logic to control what users can access and do within your application. Continue to the Authorization section to learn more.

Creating a Session on the Server:

```
import db from '../../lib/db'
import type { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  try {
    const user = req.body
    const sessionId = generateSessionId()
    await db.insertSession({
      sessionId,
      userId: user.id,
      createdAt: new Date(),
    })

    res.status(200).json({ sessionId })
  } catch (error) {
    res.status(500).json({ error: 'Internal Server Error' })
  }
}
```

```
import db from '../../lib/db'

export default async function handler(req, res) {
  try {
    const user = req.body
    const sessionId = generateSessionId()
    await db.insertSession({
      sessionId,
      userId: user.id,
      createdAt: new Date(),
    })

    res.status(200).json({ sessionId })
  } catch (error) {
    res.status(500).json({ error: 'Internal Server Error' })
  }
}
```

Authorization

Once a user is authenticated and a session is created, you can implement authorization to control what the user can access and do within your application.

There are two main types of authorization checks:

1. **Optimistic**: Checks if the user is authorized to access a route or perform an action using the session data stored in the cookie. These checks are useful for quick operations, such as showing/hiding UI elements or redirecting users based on permissions or roles.
2. **Secure**: Checks if the user is authorized to access a route or perform an action using the session data stored in the database. These checks are more secure and are used for operations that require access to sensitive data or actions.

For both cases, we recommend:

- Creating a Data Access Layer to centralize your authorization logic
- Using Data Transfer Objects (DTO) to only return the necessary data
- Optionally use Middleware to perform optimistic checks.

Optimistic checks with Middleware (Optional)

There are some cases where you may want to use Middleware and redirect users based on permissions:

- To perform optimistic checks. Since Middleware runs on every route, it's a good way to centralize redirect logic and pre-filter unauthorized users.
- To protect static routes that share data between users (e.g. content behind a paywall).

However, since Middleware runs on every route, including prefetched routes, it's important to only read the session from the cookie (optimistic checks), and avoid database checks to prevent performance issues.

For example:

```
import { NextRequest, NextResponse } from 'next/server'
import { decrypt } from '@/app/lib/session'
import { cookies } from 'next/headers'

// 1. Specify protected and public routes
const protectedRoutes = ['/dashboard']
const publicRoutes = ['/login', '/signup', '/']

export default async function middleware(req: NextRequest) {
  // 2. Check if the current route is protected or public
  const path = req.nextUrl.pathname
  const isProtectedRoute = protectedRoutes.includes(path)
  const isPublicRoute = publicRoutes.includes(path)

  // 3. Decrypt the session from the cookie
  const cookie = (await cookies()).get('session')?.value
  const session = await decrypt(cookie)

  // 4. Redirect to /login if the user is not authenticated
  if (isProtectedRoute && !session?.userId) {
    return NextResponse.redirect(new URL('/login', req.nextUrl))
  }

  // 5. Redirect to /dashboard if the user is authenticated
  if (
    isPublicRoute &&
    session?.userId &&
    !req.nextUrl.pathname.startsWith('/dashboard')
```

```
  ) {
    return NextResponse.redirect(new URL('/dashboard', req.nextUrl))
  }

  return NextResponse.next()
}

// Routes Middleware should not run on
export const config = {
  matcher: ['/((?!api|_next/static|_next/image|.*\\.png$).*)'],
}
```

```
import { NextResponse } from 'next/server'
import { decrypt } from '@/app/lib/session'
import { cookies } from 'next/headers'

// 1. Specify protected and public routes
const protectedRoutes = ['/dashboard']
const publicRoutes = ['/login', '/signup', '/']

export default async function middleware(req) {
  // 2. Check if the current route is protected or public
  const path = req.nextUrl.pathname
  const isProtectedRoute = protectedRoutes.includes(path)
  const isPublicRoute = publicRoutes.includes(path)

  // 3. Decrypt the session from the cookie
  const cookie = (await cookies()).get('session')?.value
  const session = await decrypt(cookie)

  // 5. Redirect to /login if the user is not authenticated
  if (isProtectedRoute && !session?.userId) {
    return NextResponse.redirect(new URL('/login', req.nextUrl))
  }

  // 6. Redirect to /dashboard if the user is authenticated
  if (
    isPublicRoute &&
    session?.userId &&
    !req.nextUrl.pathname.startsWith('/dashboard')
  ) {
    return NextResponse.redirect(new URL('/dashboard', req.nextUrl))
  }

  return NextResponse.next()
}

// Routes Middleware should not run on
export const config = {
  matcher: ['/((?!api|_next/static|_next/image|.*\\.png$).*)'],
}
```

While Middleware can be useful for initial checks, it should not be your only line of defense in protecting your data. The majority of security checks should be performed as close as possible to your data source, see Data Access Layer for more information.

> **Tips**:
>
> - In Middleware, you can also read cookies using `req.cookies.get('session').value`.
> - Middleware uses the Edge Runtime, check if your Auth library and session management library are compatible.
> - You can use the `matcher` property in the Middleware to specify which routes Middleware should run on. Although, for auth, it's recommended Middleware runs on all routes.

Creating a Data Access Layer (DAL)

We recommend creating a DAL to centralize your data requests and authorization logic.

The DAL should include a function that verifies the user's session as they interact with your application. At the very least, the function should check if the session is valid, then redirect or return the user information needed to make further requests.

For example, create a separate file for your DAL that includes a `verifySession()` function. Then use React's cache API to memoize the return value of the function during a React render pass:

```
import 'server-only'

import { cookies } from 'next/headers'
import { decrypt } from '@/app/lib/session'

export const verifySession = cache(async () => {
  const cookie = (await cookies()).get('session')?.value
  const session = await decrypt(cookie)

  if (!session?.userId) {
    redirect('/login')
  }

  return { isAuth: true, userId: session.userId }
})
```

```
import 'server-only'

import { cookies } from 'next/headers'
import { decrypt } from '@/app/lib/session'

export const verifySession = cache(async () => {
  const cookie = (await cookies()).get('session')?.value
  const session = await decrypt(cookie)

  if (!session.userId) {
    redirect('/login')
  }

  return { isAuth: true, userId: session.userId }
})
```

You can then invoke the `verifySession()` function in your data requests, Server Actions, Route Handlers:

```
export const getUser = cache(async () => {
  const session = await verifySession()
```

```
  if (!session) return null

  try {
    const data = await db.query.users.findMany({
      where: eq(users.id, session.userId),
      // Explicitly return the columns you need rather than the whole user object
      columns: {
        id: true,
        name: true,
        email: true,
      },
    })

    const user = data[0]

    return user
  } catch (error) {
    console.log('Failed to fetch user')
    return null
  }
})

export const getUser = cache(async () => {
  const session = await verifySession()
  if (!session) return null

  try {
    const data = await db.query.users.findMany({
      where: eq(users.id, session.userId),
      // Explicitly return the columns you need rather than the whole user object
      columns: {
        id: true,
        name: true,
        email: true,
      },
    })

    const user = data[0]

    return user
  } catch (error) {
    console.log('Failed to fetch user')
    return null
  }
})
```

Tip:

- A DAL can be used to protect data fetched at request time. However, for static routes that share data between users, data will be fetched at build time and not at request time. Use Middleware to protect static routes.
- For secure checks, you can check if the session is valid by comparing the session ID with your database. Use React's cache function to avoid unnecessary duplicate requests to the database during a render pass.
- You may wish to consolidate related data requests in a JavaScript class that runs `verifySession()` before any methods.

Using Data Transfer Objects (DTO)

When retrieving data, it's recommended you return only the necessary data that will be used in your application, and not entire objects. For example, if you're fetching user data, you might only return the user's ID and name, rather than the entire user object which could contain passwords, phone numbers, etc.

However, if you have no control over the returned data structure, or are working in a team where you want to avoid whole objects being passed to the client, you can use strategies such as specifying what fields are safe to be exposed to the client.

```
import 'server-only'
import { getUser } from '@/app/lib/dal'

function canSeeUsername(viewer: User) {
  return true
}

function canSeePhoneNumber(viewer: User, team: string) {
  return viewer.isAdmin || team === viewer.team
}

export async function getProfileDTO(slug: string) {
  const data = await db.query.users.findMany({
    where: eq(users.slug, slug),
    // Return specific columns here
  })
  const user = data[0]

  const currentUser = await getUser(user.id)

  // Or return only what's specific to the query here
  return {
    username: canSeeUsername(currentUser) ? user.username : null,
    phonenumber: canSeePhoneNumber(currentUser, user.team)
      ? user.phonenumber
      : null,
  }
}
```

```
import 'server-only'
import { getUser } from '@/app/lib/dal'

function canSeeUsername(viewer) {
  return true
}

function canSeePhoneNumber(viewer, team) {
  return viewer.isAdmin || team === viewer.team
}

export async function getProfileDTO(slug) {
  const data = await db.query.users.findMany({
    where: eq(users.slug, slug),
    // Return specific columns here
  })
  const user = data[0]
```

```
  const currentUser = await getUser(user.id)

  // Or return only what's specific to the query here
  return {
    username: canSeeUsername(currentUser) ? user.username : null,
    phonenumber: canSeePhoneNumber(currentUser, user.team)
      ? user.phonenumber
      : null,
  }
}
```

By centralizing your data requests and authorization logic in a DAL and using DTOs, you can ensure that all data requests are secure and consistent, making it easier to maintain, audit, and debug as your application scales.

Good to know:

- There are a couple of different ways you can define a DTO, from using `toJSON()`, to individual functions like the example above, or JS classes. Since these are JavaScript patterns and not a React or Next.js feature, we recommend doing some research to find the best pattern for your application.
- Learn more about security best practices in our Security in Next.js article.

Server Components

Auth check in Server Components are useful for role-based access. For example, to conditionally render components based on the user's role:

```
import { verifySession } from '@/app/lib/dal'

export default function Dashboard() {
  const session = await verifySession()
  const userRole = session?.user?.role // Assuming 'role' is part of the session object

  if (userRole === 'admin') {
    return <AdminDashboard />
  } else if (userRole === 'user') {
    return <UserDashboard />
  } else {
    redirect('/login')
  }
}
```

```
import { verifySession } from '@/app/lib/dal'

export default function Dashboard() {
  const session = await verifySession()
  const userRole = session.role // Assuming 'role' is part of the session object

  if (userRole === 'admin') {
    return <AdminDashboard />
  } else if (userRole === 'user') {
    return <UserDashboard />
  } else {
    redirect('/login')
  }
}
```

In the example, we use the `verifySession()` function from our DAL to check for 'admin', 'user', and unauthorized roles. This pattern ensures that each user interacts only with components appropriate to their role.

Layouts and auth checks

Due to Partial Rendering, be cautious when doing checks in Layouts as these don't re-render on navigation, meaning the user session won't be checked on every route change.

Instead, you should do the checks close to your data source or the component that'll be conditionally rendered.

For example, consider a shared layout that fetches the user data and displays the user image in a nav. Instead of doing the auth check in the layout, you should fetch the user data (`getUser()`) in the layout and do the auth check in your DAL.

This guarantees that wherever `getUser()` is called within your application, the auth check is performed, and prevents developers forgetting to check the user is authorized to access the data.

```
export default async function Layout({
  children,
}: {
  children: React.ReactNode;
}) {
  const user = await getUser();

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

```
export default async function Layout({ children }) {
  const user = await getUser();

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

```
export const getUser = cache(async () => {
  const session = await verifySession()
  if (!session) return null

  // Get user ID from session and fetch data
})
```

```
export const getUser = cache(async () => {
  const session = await verifySession()
  if (!session) return null

  // Get user ID from session and fetch data
})
```

Good to know:

- A common pattern in SPAs is to `return null` in a layout or a top-level component if a user is not authorized. This pattern is **not recommended** since Next.js applications have multiple entry points, which will not prevent nested route segments and Server Actions from being accessed.

Server Actions

Treat Server Actions with the same security considerations as public-facing API endpoints, and verify if the user is allowed to perform a mutation.

In the example below, we check the user's role before allowing the action to proceed:

```
'use server'
import { verifySession } from '@/app/lib/dal'

export async function serverAction(formData: FormData) {
  const session = await verifySession()
  const userRole = session?.user?.role

  // Return early if user is not authorized to perform the action
  if (userRole !== 'admin') {
    return null
  }

  // Proceed with the action for authorized users
}
```

```
'use server'
import { verifySession } from '@/app/lib/dal'

export async function serverAction() {
  const session = await verifySession()
  const userRole = session.user.role

  // Return early if user is not authorized to perform the action
  if (userRole !== 'admin') {
    return null
  }

  // Proceed with the action for authorized users
}
```

Route Handlers

Treat Route Handlers with the same security considerations as public-facing API endpoints, and verify if the user is allowed to access the Route Handler.

For example:

```
import { verifySession } from '@/app/lib/dal'

export async function GET() {
  // User authentication and role verification
  const session = await verifySession()

  // Check if the user is authenticated
  if (!session) {
    // User is not authenticated
    return new Response(null, { status: 401 })
  }

  // Check if the user has the 'admin' role
  if (session.user.role !== 'admin') {
    // User is authenticated but does not have the right permissions
    return new Response(null, { status: 403 })
  }

  // Continue for authorized users
}
```

```
import { verifySession } from '@/app/lib/dal'

export async function GET() {
  // User authentication and role verification
  const session = await verifySession()

  // Check if the user is authenticated
  if (!session) {
    // User is not authenticated
    return new Response(null, { status: 401 })
  }

  // Check if the user has the 'admin' role
  if (session.user.role !== 'admin') {
    // User is authenticated but does not have the right permissions
    return new Response(null, { status: 403 })
  }

  // Continue for authorized users
}
```

The example above demonstrates a Route Handler with a two-tier security check. It first checks for an active session, and then verifies if the logged-in user is an 'admin'.

Context Providers

Using context providers for auth works due to interleaving. However, React `context` is not supported in Server Components, making them only applicable to Client Components.

This works, but any child Server Components will be rendered on the server first, and will not have access to the context provider's session data:

```
import { ContextProvider } from 'auth-lib'

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

```
"use client";

import { useSession } from "auth-lib";

export default function Profile() {
  const { userId } = useSession();
  const { data } = useSWR(`/api/user/${userId}`, fetcher)

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

```
"use client";

import { useSession } from "auth-lib";

export default function Profile() {
  const { userId } = useSession();
  const { data } = useSWR(`/api/user/${userId}`, fetcher)

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

If session data is needed in Client Components (e.g. for client-side data fetching), use React's `taintUniqueValue` API to prevent sensitive session data from being exposed to the client.

Creating a Data Access Layer (DAL)

Protecting API Routes

API Routes in Next.js are essential for handling server-side logic and data management. It's crucial to secure these routes to ensure that only authorized users can access specific functionalities. This typically involves verifying the user's authentication status and their role-based permissions.

Here's an example of securing an API Route:

```
import { NextApiRequest, NextApiResponse } from 'next'

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
) {
  const session = await getSession(req)

  // Check if the user is authenticated
  if (!session) {
    res.status(401).json({
      error: 'User is not authenticated',
    })
    return
  }

  // Check if the user has the 'admin' role
  if (session.user.role !== 'admin') {
    res.status(401).json({
      error: 'Unauthorized access: User does not have admin privileges.',
    })
    return
  }

  // Proceed with the route for authorized users
  // ... implementation of the API Route
}
```

```
export default async function handler(req, res) {
  const session = await getSession(req)

  // Check if the user is authenticated
  if (!session) {
    res.status(401).json({
      error: 'User is not authenticated',
    })
    return
  }

  // Check if the user has the 'admin' role
  if (session.user.role !== 'admin') {
    res.status(401).json({
      error: 'Unauthorized access: User does not have admin privileges.',
    })
    return
  }

  // Proceed with the route for authorized users
  // ... implementation of the API Route
}
```

This example demonstrates an API Route with a two-tier security check for authentication and authorization. It first checks for an active session, and then verifies if the logged-in user is an 'admin'. This approach ensures secure access, limited to authenticated and authorized users, maintaining robust security for request processing.

Resources

Now that you've learned about authentication in Next.js, here are Next.js-compatible libraries and resources to help you implement secure authentication and session management:

Auth Libraries

- Auth0
- Clerk
- Kinde
- NextAuth.js
- Ory
- Stack Auth
- Supabase
- Stytch

- WorkOS

Session Management Libraries

- Iron Session
- Jose

Further Reading

To continue learning about authentication and security, check out the following resources:

- How to think about security in Next.js
- Understanding XSS Attacks
- Understanding CSRF Attacks
- The Copenhagen Book

---

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 Optimization:** 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 APIs:** Be aware that Dynamic APIs 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 API 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 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 help 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.

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
- Server Actions

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: Multi-Zones description: Learn how to build micro-frontends using Next.js Multi-Zones to deploy multiple Next.js apps under a single domain.

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

▼ Examples

- With Zones

Multi-Zones are an approach to micro-frontends that separate a large application on a domain into smaller Next.js applications that each serve a set of paths. This is useful when there are collections of pages unrelated to the other pages in the application. By moving those pages to a separate zone (i.e., a separate application), you can reduce the size of each application which improves build times and removes code that is only necessary for one of the zones. Since applications are decoupled, Multi-Zones also allows other applications on the domain to use their own choice of framework.

For example, let's say you have the following set of pages that you would like to split up:

- `/blog/*` for all blog posts
- `/dashboard/*` for all pages when the user is logged-in to the dashboard
- `/*` for the rest of your website not covered by other zones

With Multi-Zones support, you can create three applications that all are served on the same domain and look the same to the user, but you can develop and deploy each of the applications independently.

Navigating between pages in the same zone will perform soft navigations, a navigation that does not require reloading the page. For example, in this diagram, navigating from `/` to `/products` will be a soft navigation.

Navigating from a page in one zone to a page in another zone, such as from `/` to `/dashboard`, will perform a hard navigation, unloading the resources of the current page and loading the resources of the new page. Pages that are frequently visited together should live in the same zone to avoid hard navigations.

How to define a zone

A zone is a normal Next.js application where you also configure an assetPrefix to avoid conflicts with pages and static files in other zones.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  assetPrefix: '/blog-static',
}
```

Next.js assets, such as JavaScript and CSS, will be prefixed with `assetPrefix` to make sure that they don't conflict with assets from other zones. These assets will be served under `/assetPrefix/_next/...` for each of the zones.

The default application handling all paths not routed to another more specific zone does not need an `assetPrefix`.

In versions older than Next.js 15, you may also need an additional rewrite to handle the static assets. This is no longer necessary in Next.js 15.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  assetPrefix: '/blog-static',
  async rewrites() {
    return {
      beforeFiles: [
        {
          source: '/blog-static/_next/:path+',
          destination: '/_next/:path+',
        },
      ],
    }
  },
}
```

How to route requests to the right zone

With the Multi Zones set-up, you need to route the paths to the correct zone since they are served by different applications. You can use any HTTP proxy to do this, but one of the Next.js applications can also be used to route requests for the entire domain.

To route to the correct zone using a Next.js application, you can use `rewrites`. For each path served by a different zone, you would add a rewrite rule to send that path to the domain of the other zone. For example:

```
async rewrites() {
    return [
        {
            source: '/blog',
            destination: `${process.env.BLOG_DOMAIN}/blog`,
        },
        {
            source: '/blog/:path+',
            destination: `${process.env.BLOG_DOMAIN}/blog/:path+`,
        }
    ];
}
```

`destination` should be a URL that is served by the zone, including scheme and domain. This should point to the zone's production domain, but it can also be used to route requests to `localhost` in local development.

> **Good to know**: URL paths should be unique to a zone. For example, two zones trying to serve `/blog` would create a routing conflict.

Routing requests using middleware

Routing requests through `rewrites` is recommended to minimize latency overhead for the requests, but middleware can also be used when there is a need for a dynamic decision when routing. For example, if you are using a feature flag to decide where a path should be routed such as during a migration, you can use middleware.

```
export async function middleware(request) {
  const { pathname, search } = req.nextUrl;
  if (pathname === '/your-path' && myFeatureFlag.isEnabled()) {
    return NextResponse.rewrite(`${rewriteDomain}${pathname}${search}});
  }
}
```

Linking between zones

Links to paths in a different zone should use an `a` tag instead of the Next.js `<Link>` component. This is because Next.js will try to prefetch and soft navigate to any relative path in `<Link>` component, which will not work across zones.

Sharing code

The Next.js applications that make up the different zones can live in any repository. However, it is often convenient to put these zones in a monorepo to more easily share code. For zones that live in different repositories, code can also be shared using public or private NPM packages.

Since the pages in different zones may be released at different times, feature flags can be useful for enabling or disabling features in unison across the different zones.

For Next.js on Vercel applications, you can use a monorepo to deploy all affected zones with a single `git push`.

Server Actions

When using Server Actions with Multi-Zones, you must explicitly allow the user-facing origin since your user facing domain may serve multiple applications. In your `next.config.js` file, add the following lines:

```
const nextConfig = {
  experimental: {
    serverActions: {
      allowedOrigins: ['your-production-domain.com'],
    },
  },
}
```

See `serverActions.allowedOrigins` for more information.

---

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

> 📹 **Watch:** Learn more about self-hosting Next.js → YouTube (45 minutes).

We have community maintained deployment examples with the following providers:

- Deno
- DigitalOcean
- Flightcontrol
- Fly.io
- GitHub Pages
- Google Cloud Run
- Railway
- Render
- SST

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:**
>
> - On glibc-based Linux systems, Image Optimization 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.

You safely read environment variables on the server during dynamic rendering.

```
import { connection } from 'next/server'

export default async function Component() {
  await connection()
  // cookies, headers, and other Dynamic APIs
  // will also opt into dynamic rendering, meaning
  // this env variable is evaluated at runtime
  const value = process.env.MY_VALUE
  // ...
}
```

```
import { connection } from 'next/server'

export default async function Component() {
  await connection()
  // cookies, headers, and other Dynamic APIs
  // will also opt into dynamic rendering, meaning
  // this env variable is evaluated at runtime
  const value = process.env.MY_VALUE
  // ...
}
```

This allows you to use a singular Docker image that can be promoted through multiple environments with different values.

> **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 (with Incremental Static Regeneration) 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 = {
  cacheHandler: require.resolve('./cache-handler.js'),
  cacheMaxMemorySize: 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(tags) {
    // tags is either a string or an array of strings
    tags = [tags].flat()
    // 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.some((tag) => tags.include(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 `deploymentId`, 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 version are still available to older clients, even after the new version is deployed.

You can manually configure the `deploymentId` property in your `next.config.js` file to ensure each request uses either `?dpl` query string or `x-deployment-id` header.

Streaming and Suspense

The Next.js App Router supports streaming responses when self-hosting. If you are using Nginx or a similar proxy, you will need to configure it to disable buffering to enable streaming.

For example, you can disable buffering in Nginx by setting `X-Accel-Buffering` to `no`:

```
module.exports = {
  async headers() {
    return [
      {
        source: '/:path*{/}?',
        headers: [
          {
            key: 'X-Accel-Buffering',
            value: 'no',
          },
        ],
      },
    ]
  },
}
```

Partial Prerendering

Partial Prerendering (experimental) works by default with Next.js and is not a CDN feature. This includes deployment as a Node.js server (through `next start`) and when used with a Docker container.

Usage with CDNs

When using a CDN in front on your Next.js application, the page will include `Cache-Control: private` response header when dynamic APIs are accessed. This ensures that the resulting HTML page is marked as non-cachable. If the page is fully prerendered to static, it will include `Cache-Control: public` to allow the page to be cached on the CDN.

If you don't need a mix of both static and dynamic components, you can make your entire route static and cache the output HTML on a CDN. This Automatic Static Optimization is the default behavior when running `next build` if dynamic APIs are not used.

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

Codemods

15.0

Transform App Router Route Segment Config `runtime` value from `experimental-edge` to `edge`

`app-dir-runtime-config-experimental-edge`

> **Note**: This codemod is App Router specific.

```
npx @next/codemod@latest app-dir-runtime-config-experimental-edge .
```

This codemod transforms Route Segment Config `runtime` value `experimental-edge` to `edge`.

For example:

```
export const runtime = 'experimental-edge'
```

Transforms into:

```
export const runtime = 'edge'
```

Migrate to async Dynamic APIs

APIs that opted into dynamic rendering that previously supported synchronous access are now asynchronous. You can read more about this breaking change in the upgrade guide.

`next-async-request-api`

```
npx @next/codemod@latest next-async-request-api .
```

This codemod will transform dynamic APIs (`cookies()`, `headers()` and `draftMode()` from `next/headers`) that are now asynchronous to be properly awaited or wrapped with `React.use()` if applicable. When an automatic migration isn't possible, the codemod will either add a typecast (if a TypeScript file) or a comment to inform the user that it needs to be manually reviewed & updated.

For example:

```
import { cookies, headers } from 'next/headers'
const token = cookies().get('token')

function useToken() {
  const token = cookies().get('token')
  return token
}

export default function Page() {
  const name = cookies().get('name')
}

function getHeader() {
  return headers().get('x-foo')
}
```

Transforms into:

```
import { use } from 'react'
import { cookies, headers, type UnsafeUnwrappedCookies } from 'next/headers'

const token = (await cookies()).get('token')

function useToken() {
  const token = use(cookies()).get('token')
  return token
}

export default function Page() {
  const name = (await cookies()).get('name')
}

function getHeader() {
  return (headers() as UnsafeUnwrappedCookies).get('x-foo')
}
```

When we detect property access on the `params` or `searchParams` props in the page / route entries (`page.js`, `layout.js`, `route.js`, or `default.js`) or the `generateMetadata` / `generateViewport` APIs, it will attempt to transform the callsite from a sync to an async function, and await the property access. If it can't be made async (such as with a client component), it will use `React.use` to unwrap the promise .

For example:

```
// page.tsx
export default function Page({
  params,
  searchParams,
}: {
  params: { slug: string }
  searchParams: { [key: string]: string | string[] | undefined }
}) {
  const { value } = searchParams
  if (value === 'foo') {
    // ...
  }
}

export function generateMetadata({ params }: { params: { slug: string } }) {
  return {
    title: `My Page - ${slug}`,
  }
}
```

Transforms into:

```
// page.tsx
export default function Page(props: {
  params: { slug: string }
  searchParams: { [key: string]: string | string[] | undefined }
}) {
  const { value } = await props.searchParams
  if (value === 'foo') {
    // ...
  }
}

export function generateMetadata(props: { params: { slug: string } }) {
  const { slug } = await props.params
  return {
    title: `My Page - ${slug}`,
  }
}
```

> **Good to know:** When this codemod identifies a spot that might require manual intervention, but we aren't able to determine the exact fix, it will add a comment or typecast to the code to inform the user that it needs to be manually updated. These comments are prefixed with **@next/codemod**, and typecasts are prefixed with `UnsafeUnwrapped`. Your build will error until these comments are explicitly removed. Read more.

Replace `geo` and `ip` properties of `NextRequest` with `@vercel/functions`

`next-request-geo-ip`

```
npx @next/codemod@latest next-request-geo-ip .
```

This codemod installs `@vercel/functions` and transforms `geo` and `ip` properties of `NextRequest` with corresponding `@vercel/functions` features.

For example:

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

export function GET(req: NextRequest) {
  const { geo, ip } = req
}
```

Transforms into:

```
import type { NextRequest } from 'next/server'
import { geolocation, ipAddress } from '@vercel/functions'

export function GET(req: NextRequest) {
  const geo = geolocation(req)
  const ip = ipAddress(req)
}
```

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: Version 15 description: Upgrade your Next.js Application from Version 14 to 15.

{/* 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 14 to 15

To update to Next.js version 15, you can use the `upgrade` codemod:

```
npx @next/codemod@canary upgrade latest
```

If you prefer to do it manually, ensure that you're installing the latest Next & React RC, e.g.:

```
npm i next@latest react@rc react-dom@rc eslint-config-next@latest
```

> **Good to know:**
>
> - If you see a peer dependencies warning, you may need to update `react` and `react-dom` to the suggested versions, or you use the `--force` or `--legacy-peer-deps` flag to ignore the warning. This won't be necessary once both Next.js 15 and React 19 are stable.
> - If you are using TypeScript, you'll need to temporarily override the React types. See the React 19 RC upgrade guide for more information.

React 19

- The minimum versions of `react` and `react-dom` is now 19.
- `useFormState` has been replaced by `useActionState`. The `useFormState` hook is still available in React 19, but it is deprecated and will be removed in a future release. `useActionState` is recommended and includes additional properties like reading the `pending` state directly. Learn more.
- `useFormStatus` now includes additional keys like `data`, `method`, and `action`. If you are not using React 19, only the `pending` key is available. Learn more.
- Read more in the React 19 upgrade guide.

Async Request APIs (Breaking change)

Previously synchronous Dynamic APIs that rely on runtime information are now **asynchronous**:

- `cookies`
- `headers`
- `draftMode`
- `params` in `layout.js`, `page.js`, `route.js`, `default.js`, `opengraph-image`, `twitter-image`, `icon`, and `apple-icon`.

- `searchParams` in `page.js`

To ease the burden of migration, a codemod is available to automate the process and the APIs can temporarily be accessed synchronously.

`cookies`

Recommended Async Usage

```
import { cookies } from 'next/headers'

// Before
const cookieStore = cookies()
const token = cookieStore.get('token')

// After
const cookieStore = await cookies()
const token = cookieStore.get('token')
```

Temporary Synchronous Usage

```
import { cookies, type UnsafeUnwrappedCookies } from 'next/headers'

// Before
const cookieStore = cookies()
const token = cookieStore.get('token')

// After
const cookieStore = cookies() as unknown as UnsafeUnwrappedCookies
// will log a warning in dev
const token = cookieStore.get('token')
```

```
import { cookies } from 'next/headers'

// Before
const cookieStore = cookies()
const token = cookieStore.get('token')

// After
const cookieStore = cookies()
// will log a warning in dev
const token = cookieStore.get('token')
```

`headers`

Recommended Async Usage

```
import { headers } from 'next/headers'

// Before
const headersList = headers()
const userAgent = headersList.get('user-agent')

// After
const headersList = await headers()
const userAgent = headersList.get('user-agent')
```

Temporary Synchronous Usage

```
import { headers, type UnsafeUnwrappedHeaders } from 'next/headers'

// Before
const headersList = headers()
const userAgent = headersList.get('user-agent')

// After
const headersList = headers() as unknown as UnsafeUnwrappedHeaders
// will log a warning in dev
const userAgent = headersList.get('user-agent')
```

```
import { headers } from 'next/headers'

// Before
const headersList = headers()
const userAgent = headersList.get('user-agent')

// After
const headersList = headers()
// will log a warning in dev
const userAgent = headersList.get('user-agent')
```

`draftMode`

Recommended Async Usage

```
import { draftMode } from 'next/headers'

// Before
const { isEnabled } = draftMode()

// After
const { isEnabled } = await draftMode()
```

Temporary Synchronous Usage

```
import { draftMode, type UnsafeUnwrappedDraftMode } from 'next/headers'

// Before
const { isEnabled } = draftMode()

// After
// will log a warning in dev
const { isEnabled } = draftMode() as unknown as UnsafeUnwrappedDraftMode
```

```
import { draftMode } from 'next/headers'

// Before
const { isEnabled } = draftMode()

// After
// will log a warning in dev
const { isEnabled } = draftMode()
```

`params` & `searchParams`

Asynchronous Layout

```
// Before
type Params = { slug: string }

export function generateMetadata({ params }: { params: Params }) {
  const { slug } = params
}

export default async function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Params
}) {
  const { slug } = params
}

// After
type Params = Promise<{ slug: string }>

export async function generateMetadata({ params }: { params: Params }) {
  const { slug } = await params
}

export default async function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Params
}) {
  const { slug } = await params
}
```

```
// Before
export function generateMetadata({ params }) {
  const { slug } = params
}

export default async function Layout({ children, params }) {
  const { slug } = params
}

// After
export async function generateMetadata({ params }) {
  const { slug } = await params
}

export default async function Layout({ children, params }) {
  const { slug } = await params
}
```

Synchronous Layout

```
// Before
type Params = { slug: string }

export default function Layout({
  children,
  params,
}: {
  children: React.ReactNode
  params: Params
}) {
  const { slug } = params
}

// After
import { use } from 'react'

type Params = Promise<{ slug: string }>

export default function Layout(props: {
  children: React.ReactNode
  params: Params
}) {
  const params = use(props.params)
  const slug = params.slug
}
```

```
// Before
export default function Layout({ children, params }) {
  const { slug } = params
}

// After
import { use } from 'react'
export default async function Layout(props) {
  const params = use(props.params)
  const slug = params.slug
}
```

Asynchronous Page

```
// Before
type Params = { slug: string }
type SearchParams = { [key: string]: string | string[] | undefined }

export function generateMetadata({
  params,
  searchParams,
}: {
  params: Params
  searchParams: SearchParams
}) {
  const { slug } = params
  const { query } = searchParams
}

export default async function Page({
  params,
  searchParams,
}: {
  params: Params
  searchParams: SearchParams
}) {
  const { slug } = params
  const { query } = searchParams
}

// After
type Params = Promise<{ slug: string }>
type SearchParams = Promise<{ [key: string]: string | string[] | undefined }>

export async function generateMetadata(props: {
  params: Params
  searchParams: SearchParams
}) {
  const params = await props.params
  const searchParams = await props.searchParams
  const slug = params.slug
  const query = searchParams.query
}

export default async function Page(props: {
  params: Params
  searchParams: SearchParams
}) {
  const params = await props.params
  const searchParams = await props.searchParams
  const slug = params.slug
  const query = searchParams.query
}

// Before
export function generateMetadata({ params, searchParams }) {
  const { slug } = params
  const { query } = searchParams
}

export default function Page({ params, searchParams }) {
  const { slug } = params
  const { query } = searchParams
}

// After
export async function generateMetadata(props) {
  const params = await props.params
  const searchParams = await props.searchParams
  const slug = params.slug
  const query = searchParams.query
}

export async function Page(props) {
  const params = await props.params
  const searchParams = await props.searchParams
  const slug = params.slug
  const query = searchParams.query
}
```

Synchronous Page

```
'use client'

// Before
type Params = { slug: string }
type SearchParams = { [key: string]: string | string[] | undefined }

export default function Page({
  params,
  searchParams,
}: {
  params: Params
  searchParams: SearchParams
}) {
  const { slug } = params
  const { query } = searchParams
}

// After
import { use } from 'react'

type Params = Promise<{ slug: string }>
type SearchParams = Promise<{ [key: string]: string | string[] | undefined }>

export default function Page(props: {
  params: Params
  searchParams: SearchParams
}) {
  const params = use(props.params)
  const searchParams = use(props.searchParams)
  const slug = params.slug
  const query = searchParams.query
}
```

```
// Before
export default function Page({ params, searchParams }) {
  const { slug } = params
  const { query } = searchParams
}

// After
import { use } from "react"

export default function Page(props) {
  const params = use(props.params)
  const searchParams = use(props.searchParams)
  const slug = params.slug
  const query = searchParams.query
}
```

Route Handlers

```
// Before
type Params = { slug: string }

export async function GET(request: Request, segmentData: { params: Params }) {
  const params = segmentData.params
  const slug = params.slug
}

// After
type Params = Promise<{ slug: string }>

export async function GET(request: Request, segmentData: { params: Params }) {
  const params = await segmentData.params
  const slug = params.slug
}
```

```
// Before
export async function GET(request, segmentData) {
  const params = segmentData.params
  const slug = params.slug
}

// After
export async function GET(request, segmentData) {
  const params = await segmentData.params
  const slug = params.slug
}
```

`runtime` configuration (Breaking change)

The `runtime` segment configuration previously supported a value of `experimental-edge` in addition to `edge`. Both configurations refer to the same thing, and to simplify the options, we will now error if `experimental-edge` is used. To fix this, update your `runtime` configuration to `edge`. A codemod is available to automatically do this.

`fetch` requests

`fetch` requests are no longer cached by default.

To opt specific `fetch` requests into caching, you can pass the `cache: 'force-cache'` option.

```
export default async function RootLayout() {
  const a = await fetch('https://...') // Not Cached
  const b = await fetch('https://...', { cache: 'force-cache' }) // Cached

  // ...
}
```

To opt all `fetch` requests in a layout or page into caching, you can use the `export const fetchCache = 'default-cache'` segment config option. If individual `fetch` requests specify a `cache` option, that will be used instead.

```
// Since this is the root layout, all fetch requests in the app
// that don't set their own cache option will be cached.
export const fetchCache = 'default-cache'

export default async function RootLayout() {
  const a = await fetch('https://...') // Cached
  const b = await fetch('https://...', { cache: 'no-store' }) // Not cached

  // ...
}
```

Route Handlers

`GET` functions in Route Handlers are no longer cached by default. To opt `GET` methods into caching, you can use a route config option such as `export const dynamic = 'force-static'` in your Route Handler file.

```
export const dynamic = 'force-static'

export async function GET() {}
```

Client-side Router Cache

When navigating between pages via `<Link>` or `useRouter`, page segments are no longer reused from the client-side router cache. However, they are still reused during browser backward and forward navigation and for shared layouts.

To opt page segments into caching, you can use the `staleTimes` config option:

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    staleTimes: {
      dynamic: 30,
      static: 180,
    },
  },
}
```

```
module.exports = nextConfig
```

Layouts and loading states are still cached and reused on navigation.

`next/font`

The `@next/font` package has been removed in favor of the built-in `next/font`. A codemod is available to safely and automatically rename your imports.

```
// Before
import { Inter } from '@next/font/google'

// After
import { Inter } from 'next/font/google'
```

bundlePagesRouterDependencies

`experimental.bundlePagesExternals` is now stable and renamed to `bundlePagesRouterDependencies`.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  // Before
  experimental: {
    bundlePagesExternals: true,
  },

  // After
  bundlePagesRouterDependencies: true,
}

module.exports = nextConfig
```

serverExternalPackages

`experimental.serverComponentsExternalPackages` is now stable and renamed to `serverExternalPackages`.

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  // Before
  experimental: {
    serverComponentsExternalPackages: ['package-name'],
  },

  // After
  serverExternalPackages: ['package-name'],
}

module.exports = nextConfig
```

Speed Insights

Auto instrumentation for Speed Insights was removed in Next.js 15.

To continue using Speed Insights, follow the Vercel Speed Insights Quickstart guide.

`NextRequest` Geolocation

The `geo` and `ip` properties on `NextRequest` have been removed as these values are provided by your hosting provider. A codemod is available to automate this migration.

If you are using Vercel, you can alternatively use the `geolocation` and `ipAddress` functions from [`@vercel/functions](https://vercel.com/docs/functions/vercel-functions-package) instead:

```
import { geolocation } from '@vercel/functions'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  const { city } = geolocation(request)

  // ...
}
```

```
import { ipAddress } from '@vercel/functions'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  const ip = ipAddress(request)

  // ...
}
```

---

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 has been removed in favor of `output: 'export'` config. 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: 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/*` API Routes have been replaced with the `route.js` (Route Handler) special file.

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 type { 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 type { 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).
  - Similar to the Pages Router, there is an optimization step to prerender Client Components to static HTML on the initial page load.
- Move the default exported page component from `pages/index.js` to `app/home-page.tsx`.

```
'use client'

// This is a Client Component (same as components in the `pages` directory)
// It receives data as props, has access to state and effects, and is
// prerendered on the server during the initial page load.
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. Search parameters and dynamic route parameters are now separate. Use the `useSearchParams` and `useParams` hooks 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.
- `route` has been removed. `usePathname` or `useSelectedLayoutSegments()` provide an alternative.

View the `useRouter()` API reference.

Sharing components between `pages` and `app`

To keep components compatible between the `pages` and `app` routers, refer to the `useRouter` hook from `next/compat/router`. This is the `useRouter` hook from the `pages` directory, but intended to be used while sharing components between routers. Once you are ready to use it only on the `app` router, update to the new `useRouter` from `next/navigation`.

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 Router, 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 = (await 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 = (await cookies()).get('theme')
  const data = await getData()
  return '...'
}
```


```
// `app` directory
import { cookies, headers } from 'next/headers'

async function getData() {
  const authHeader = (await 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 = (await 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: Migrating from Create React App description: Learn how to migrate your existing React application from Create React App to Next.js.

This guide will help you migrate an existing Create React App site to Next.js.

Why Switch?

There are several reasons why you might want to switch from Create React App to Next.js:

Slow initial page loading time

Create React App uses purely client-side React. 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 data.
2. Your application code grows with every new feature and 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.

While fetching data on the client is supported with Next.js, it also gives you the option to shift data fetching to the server, which can eliminate client-server waterfalls.

Fast and intentional loading states

With built-in support for streaming through React Suspense, 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 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: './build', // Changes the build output directory to `./dist`.
}

export default nextConfig
```

Step 3: 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 CRA 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 '...'
}
```

```
export default function RootLayout({ children }) {
  return '...'
}
```

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

Copy the content of your `index.html` file into the previously created `<RootLayout>` component while replacing the `body.div#root` and `body.noscript` 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" href="%PUBLIC_URL%/favicon.ico" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>React App</title>
        <meta name="description" content="Web site created..." />
      </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" href="%PUBLIC_URL%/favicon.ico" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>React App</title>
        <meta name="description" content="Web site created..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

> **Good to know**: Next.js ignores CRA's `public/manifest.json` file, additional iconography (except `favicon.ico`, `icon`, and `apple-icon`), and testing configuration, but if these are requirements, Next.js also supports these options. See the Metadata API and Testing docs for more information.

Step 4: Metadata

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" href="%PUBLIC_URL%/favicon.ico" />
        <title>React App</title>
        <meta name="description" content="Web site created..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <head>
        <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
        <title>React App</title>
        <meta name="description" content="Web site created..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

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>React App</title>
        <meta name="description" content="Web site created..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

```
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <head>
        <title>React App</title>
        <meta name="description" content="Web site created..." />
      </head>
      <body>
        <div id="root">{children}</div>
      </body>
    </html>
  )
}
```

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: 'React App',
  description: 'Web site created with Next.js.',
}

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

```
export const metadata = {
  title: 'React App',
  description: 'Web site created with Next.js.',
}

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: Styles

Like Create React App, Next.js has built-in support for CSS Modules.

If you're using a global CSS file, import it into your `app/layout.tsx` file:

```
import '../index.css'

// ...
```

If you're using Tailwind, you'll need to install `postcss` and `autoprefixer`:

```
npm install postcss autoprefixer
```

Then, create a `postcss.config.js` file at the root of your project:

```
module.exports = {
  plugins: {
    tailwindcss: {},
    autoprefixer: {},
  },
}
```

Step 6: 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 CRA is your `src/index.tsx` file. In this step, you'll set up the entry point of your application.

Create a `[[...slug]]` directory in your `app` directory.

Since this guide is aiming to first set up our Next.js as an SPA (Single Page Application), you need your page entry point 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.

Create a new `page.tsx` file inside the `app/[[...slug]]` directory with the following content:

```
export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return '...' // We'll update this
}
```

```
export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
```

```
  return '...' // We'll update this
}
```

This file is a Server Component. When you run `next build`, the file is prerendered into a static asset. It does *not* require any dynamic code.

This file imports our global CSS and tells `generateStaticParams` we are only going to generate one route, the index route at `/`.

Now, let's move the rest of our CRA application which will run client-only.

```
'use client'

import dynamic from 'next/dynamic'

const App = dynamic(() => import('../../App'), { ssr: false })

export function ClientOnly() {
  return <App />
}
```


```
'use client'

import dynamic from 'next/dynamic'

const App = dynamic(() => import('../../App'), { ssr: false })

export function ClientOnly() {
  return <App />
}
```


This file is a Client Component, defined by the `'use client'` directive. Client Components are still prerendered to HTML on the server before being sent to the client.

Since we want a client-only application to start, we can configure Next.js to disable prerendering from the `App` component down.

```
const App = dynamic(() => import('../../App'), { ssr: false })
```

Now, update your entrypoint page to use the new component:

```
import { ClientOnly } from './client'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return <ClientOnly />
}
```


```
import { ClientOnly } from './client'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return <ClientOnly />
}
```


Step 7: Update Static Image Imports

Next.js handles static image imports slightly different from CRA. With CRA, importing an image file will return its public URL as a string:

```
import image from './img.png'

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. You can then optionally later migrate to the `<Image>` component to take advantage of optimizing images by configuring a loader, or moving to the default Next.js server which has automatic image optimization.

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'
```

Pass the image `src` property instead of the whole image object to your `<img>` tag:

```
// Before
<img src={logo} />

// After
<img src={logo.src} />
```

Alternatively, you can reference the public URL for the image asset based on the filename. For example, `public/logo.png` will serve the image at `/logo.png` for your application, which would be the `src` value.

> **Warning:** If you're using TypeScript, you might encounter type errors when accessing the `src` property. To fix them, you need to add `next-env.d.ts` to the `include` array of your `tsconfig.json` file. Next.js will automatically generate this file when you run your application on step 9.

Step 8: Migrate the Environment Variables

Next.js has support for `.env` environment variables similar to CRA.

The main difference is the prefix used to expose environment variables on the client-side. Change all environment variables with the `REACT_APP_` prefix to `NEXT_PUBLIC_`.

Step 9: 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` file:

```
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "npx serve@latest ./build"
  }
}
```

```
# ...
.next
next-env.d.ts
```

Now run `npm run dev`, and open `http://localhost:3000`. You should see your application now running on Next.js.

Step 10: Clean Up

You can now clean up your codebase from Create React App related artifacts:

- Delete `public/index.html`
- Delete `src/index.tsx`
- Delete `src/react-app-env.d.ts`
- Delete `reportWebVitals` setup
- Uninstall CRA dependencies (`react-scripts`)

Bundler Compatibility

Create React App and Next.js both default to using webpack for bundling.

When migrating your CRA application to Next.js, you might have a custom webpack configuration you're looking to migrate. Next.js supports providing a custom webpack configuration.

Further, Next.js has support for Turbopack through `next dev --turbopack` to improve your local dev performance. Turbopack supports some webpack loaders as well for compatibility and incremental adoption.

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

  Good to know: Using a static export does not currently support using the `useParams` hook.

---

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.

While fetching data on the client is supported with Next.js, it also gives you the option to shift data fetching to the server, which can eliminate client-server waterfalls.

Fast and intentional loading states

With built-in support for streaming through React Suspense, 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 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 '...'
}

export default function RootLayout({ children }) {
  return '...'
}
```

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:**

```
import '../../index.css'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return '...' // We'll update this
}
```

```
import '../../index.css'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return '...' // We'll update this
}
```

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

This file is a Server Component. When you run `next build`, the file is prerendered into a static asset. It does *not* require any dynamic code.

This file imports our global CSS and tells `generateStaticParams` we are only going to generate one route, the index route at `/`.

Now, let's move the rest of our Vite application which will run client-only.

```
'use client'

import React from 'react'
import dynamic from 'next/dynamic'

const App = dynamic(() => import('../../App'), { ssr: false })

export function ClientOnly() {
```

```
  return <App />
}
```

```
'use client'

import React from 'react'
import dynamic from 'next/dynamic'

const App = dynamic(() => import('../../App'), { ssr: false })

export function ClientOnly() {
  return <App />
}
```

This file is a Client Component, defined by the `'use client'` directive. Client Components are still prerendered to HTML on the server before being sent to the client.

Since we want a client-only application to start, we can configure Next.js to disable prerendering from the `App` component down.

```
const App = dynamic(() => import('../../App'), { ssr: false })
```

Now, update your entrypoint page to use the new component:

```
import '../../index.css'
import { ClientOnly } from './client'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return <ClientOnly />
}
```

```
import '../../index.css'
import { ClientOnly } from './client'

export function generateStaticParams() {
  return [{ slug: [''] }]
}

export default function Page() {
  return <ClientOnly />
}
```

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. You can then optionally later migrate to the `<Image>` component to take advantage of optimizing images by configuring a loader, or moving to the default Next.js server which has automatic image optimization.

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} />
```

Alternatively, you can reference the public URL for the image asset based on the filename. For example, `public/logo.png` will serve the image at `/logo.png` for your application, which would be the `src` value.

> **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
dist
```

Now run `npm run dev`, and open `http://localhost:3000`. You should see your application now running on Next.js.

> **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: Next.js Examples nav_title: Examples description: Examples of popular Next.js UI patterns and use cases.

{/* 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. */}

Data Fetching

- Using the `fetch` API
- Using an ORM or database client
- Reading search params on the server
- Reading search params on the client

Revalidating Data

- Using ISR to revalidate data after a certain time
- Using ISR to revalidate data on-demand

Forms

- Showing a pending state while submitting a form
- Server-side form validation
- Handling expected errors
- Handling unexpected exceptions
- Showing optimistic UI updates
- Programmatic form submission

Server Actions

- Passing additional values
- Revalidating data
- Redirecting
- Setting cookies
- Deleting cookies

Metadata

- Creating an RSS feed
- Creating an Open Graph image
- Creating a sitemap
- Creating a robots.txt file
- Creating a custom 404 page
- Creating a custom 500 page

Auth

- Creating a sign-up form
- Stateless, cookie-based session management
- Stateful, database-backed session management
- Managing authorization

Testing

- Vitest
- Jest
- Playwright
- Cypress

Deployment

- Creating a Dockerfile
- Creating a static export (SPA)
- Configuring caching when self-hosting
- Configuring Image Optimization when self-hosting

---

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: Directives description: Directives are used to modify the behavior of your Next.js application.

The following directives are available:

---

title: use cache description: Learn how to use the use cache directive to cache data in your Next.js application. version: canary related: title: Related description: View related API references. links: - app/api-reference/next-config-js/dynamicIO - app/api-reference/next-config-js/cacheLife - app/api-reference/functions/cacheTag - app/api-reference/functions/cacheLife - app/api-reference/functions/revalidateTag

The `use cache` directive designates a component and/or a function to be cached. It can be used at the top of a file to indicate that all exports in the file are cacheable, or inline at the top of a function or component to inform Next.js the return value should be cached and reused for subsequent requests. This is an experimental Next.js feature, and not a native React feature like `use client` or `use server`.

Usage

Enable support for the `use cache` directive with the `dynamicIO` flag in your `next.config.ts` file:

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

const nextConfig: NextConfig = {
  experimental: {
    dynamicIO: true,
  },
}

export default nextConfig
```

Then, you can use the `use cache` directive at the file, component, or function level:

```
// File level
'use cache'
```

```
export default async function Page() {
  // ...
}

// Component level
export async function MyComponent() {
  'use cache'
  return <></>
}

// Function level
export async function getData() {
  'use cache'
  const data = await fetch('/api/data')
  return data
}
```

Good to know

- `use cache` is an experimental Next.js feature, and not a native React feature like `use client` or `use server`.
- The arguments (or props) passed to a cached function must be serializable by React. This means they can be converted to a format like JSON. Non-serializable props like functions or React elements cannot be used as arguments.
- The return value of the cacheable function must also be serializable. This ensures that the cached data can be stored and retrieved correctly.
- Functions that use the `use cache` directive must not have any side-effects, such as modifying state, directly manipulating the DOM, or setting timers to execute code at intervals.
- If used alongside Partial Prerendering, segments that have `use cache` will be prerendered as part of the static HTML shell.
- The `use cache` directive will be available separately from the `dynamicIO` flag in the future.
- Unlike `unstable_cache` which only supports JSON data, `use cache` can cache any serializable data React can render, including the render output of components.

Examples

Caching entire routes with `use cache`

To prerender an entire route, add `use cache` to the top **both** the `layout` and `page` files. Each of these segments are treated as separate entry points in your application, and will be cached independently.

```
"use cache"
import { unstable_cacheLife as cacheLife } from 'next/cache'

export default Layout({children}: {children: ReactNode}) {
  return <div>{children}</div>
}
```

```
"use cache"
import { unstable_cacheLife as cacheLife } from 'next/cache'

export default Layout({ children }) {
  return <div>{children}</div>
}
```

Any components imported and nested in `page` file will inherit the cache behavior of `page`.

```
"use cache"
import { unstable_cacheLife as cacheLife } from 'next/cache'

async function Users() {
  const users = await fetch('/api/users');
  // loop through users
}

export default Page() {
  return (
    <main>
      <Users/>
    </main>
  )
}
```

```
"use cache"
import { unstable_cacheLife as cacheLife } from 'next/cache'

async function Users() {
  const users = await fetch('/api/users');
  // loop through users
}

export default Page() {
  return (
    <main>
      <Users/>
    </main>
  )
}
```

This is recommended for applications that previously used the `export const dynamic = "force-static"` option, and will ensure the entire route is prerendered.

Caching component output with `use cache`

You can use `use cache` at the component level to cache any fetches or computations performed within that component. When you reuse the component throughout your application it can share the same cache entry as long as the props maintain the same structure.

The props are serialized and form part of the cache key, and the cache entry will be reused as long as the serialized props produce the same value in each instance.

```
export async function Bookings({ type = 'haircut' }: BookingsProps) {
  'use cache'
  async function getBookingsData() {
    const data = await fetch(`/api/bookings?type=${encodeURIComponent(type)}`)
    return data
  }
  return //...
}

interface BookingsProps {
  type: string
}
```

```
export async function Bookings({ type = 'haircut' }) {
  'use cache'
  async function getBookingsData() {
    const data = await fetch(`/api/bookings?type=${encodeURIComponent(type)}`)
    return data
  }
  return //...
}
```

Caching function output with `use cache`

Since you can add `use cache` to any asynchronous function, you aren't limited to caching components or routes only. You might want to cache a network request or database query or compute something that is very slow. By adding `use cache` to a function containing this type of work it becomes cacheable, and when reused, will share the same cache entry.

```
export async function getData() {
  'use cache'

  const data = await fetch('/api/data')
  return data
}
```

```
export async function getData() {
  'use cache'

  const data = await fetch('/api/data')
  return data
}
```

Revalidating

By default, Next.js sets a **revalidation period** **of 15 minutes** when you use the `use cache` directive. Next.js sets a near-infinite expiration duration, meaning it's suitable for content that doesn't need frequent updates.

While this revalidation period may be useful for content you don't expect to change often, you can use the `cacheLife` and `cacheTag` APIs to configure the cache behavior:

- `cacheLife`: For time-based revalidation periods.
- `cacheTag`: For on-demand revalidation.

Both of these APIs integrate across the client and server caching layers, meaning you can configure your caching semantics in one place and have them apply everywhere.

See the `cacheLife` and `cacheTag` docs for more information.

Interleaving

If you need to pass non-serializable arguments to a cacheable function, you can pass them as `children`. This means the `children` reference can change without affecting the cache entry.

```
async function Page() {
  const uncachedData = await getData()
  return (
    <CacheComponent>
      <DynamicComponent data={uncachedData} />
    </CacheComponent>
  )
}

async function CacheComponent({ children }) {
  'use cache'
  const cachedData = await fetch('/api/cached-data')
  return (
    <div>
      <PrerenderedComponent data={cachedData} />
      {children}
    </div>
  )
}
```

You can also pass Server Actions through cached components to Client Components without invoking them inside the cacheable function.

```
import ClientComponent from './ClientComponent';

async function Page() {
  const performUpdate = async () => {
    "use server"
    // Perform some server-side update
    await db.update(...);
  };

  return <CacheComponent performUpdate={performUpdate} />;
}

async function CachedComponent({ performUpdate }) {
  "use cache"
  // Do not call performUpdate here
  return <ClientComponent action={performUpdate} />;
}
```

```
import ClientComponent from './ClientComponent';

async function Page() {
  const performUpdate = async () => {
    "use server"
    // Perform some server-side update
    await db.update(...);
  };

  return <CacheComponent performUpdate={performUpdate} />;
}

async function CachedComponent({ performUpdate }) {
  "use cache"
  // Do not call performUpdate here
  return <ClientComponent action={performUpdate} />;
}
```

```
'use client'

export default function ClientComponent({ action }) {
```

```
  return <button onClick={action}>Update</button>
}
```

```
'use client'

export default function ClientComponent({ action }) {
  return <button onClick={action}>Update</button>
}
```

---

title: use client description: Learn how to use the use client directive to render a component on the client.

The `use client` directive designates a component to be rendered on the **client side** and should be used when creating interactive user interfaces (UI) that require client-side JavaScript capabilities, such as state management, event handling, and access to browser APIs. This is a React feature.

Usage

To designate a component as a Client Component, add the `use client` directive **at the top of the file**, before any imports:

```
'use client'

import { useState } from 'react'

export default function Counter() {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  )
}
```

```
'use client'

import { useState } from 'react'

export default function Counter() {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  )
}
```

Nesting Client Components within Server Components

Combining Server and Client Components allows you to build applications that are both performant and interactive:

1. **Server Components**: Use for static content, data fetching, and SEO-friendly elements.
2. **Client Components**: Use for interactive elements that require state, effects, or browser APIs.
3. **Component composition**: Nest Client Components within Server Components as needed for a clear separation of server and client logic.

In the following example:

- `Header` is a Server Component handling static content.
- `Counter` is a Client Component enabling interactivity within the page.

```
import Header from './header'
import Counter from './counter' // This is a Client Component

export default function Page() {
  return (
    <div>
      <Header />
      <Counter />
    </div>
  )
}
```

```
import Header from './header'
import Counter from './counter' // This is a Client Component

export default function Page() {
  return (
    <div>
      <Header />
      <Counter />
    </div>
  )
}
```

Reference

See the React documentation for more information on `use client`.

---

title: use server description: Learn how to use the use server directive to execute code on the server.

The `use server` directive designates a function or file to be executed on the **server side**. It can be used at the top of a file to indicate that all functions in the file are server-side, or inline at the top of a function to mark the function as a Server Function. This is a React feature.

Using `use server` at the top of a file

The following example shows a file with a `use server` directive at the top. All functions in the file are executed on the server.

```
'use server'
import { db } from '@/lib/db' // Your database client

export async function createUser(data: { name: string; email: string }) {
  const user = await db.user.create({ data })
  return user
}
```

```
'use server'
import { db } from '@/lib/db' // Your database client

export async function createUser(data) {
  const user = await db.user.create({ data })
  return user
}
```

Using Server Functions in a Client Component

To use Server Functions in Client Components you need to create your Server Functions in a dedicated file using the `use server` directive at the top of the file. These Server Functions can then be imported into Client and Server Components and executed.

Assuming you have a `fetchUsers` Server Function in `actions.ts`:

```
'use server'
import { db } from '@/lib/db' // Your database client

export async function fetchUsers() {
  const users = await db.user.findMany()
  return users
}
```

```
'use server'
import { db } from '@/lib/db' // Your database client

export async function fetchUsers() {
  const users = await db.user.findMany()
  return users
}
```

Then you can import the `fetchUsers` Server Function into a Client Component and execute it on the client-side.

```
'use client'
import { fetchUsers } from '../actions'

export default function MyButton() {
  return <button onClick={() => fetchUsers()}>Fetch Users</button>
}
```

```
'use client'
import { fetchUsers } from '../actions'

export default function MyButton() {
  return <button onClick={() => fetchUsers()}>Fetch Users</button>
}
```

Using `use server` inline

In the following example, `use server` is used inline at the top of a function to mark it as a Server Function:

```
import { db } from '@/lib/db' // Your database client

export default function UserList() {
  async function fetchUsers() {
    'use server'
    const users = await db.user.findMany()
    return users
  }

  return <button onClick={() => fetchUsers()}>Fetch Users</button>
}
```

```
import { db } from '@/lib/db' // Your database client

export default function UserList() {
  async function fetchUsers() {
    'use server'
    const users = await db.user.findMany()
    return users
  }

  return <button onClick={() => fetchUsers()}>Fetch Users</button>
}
```

Security considerations

When using the `use server` directive, it's important to ensure that all server-side logic is secure and that sensitive data remains protected.

Authentication and authorization

Always authenticate and authorize users before performing sensitive server-side operations.

```
'use server'

import { db } from '@/lib/db' // Your database client
import { authenticate } from '@/lib/auth' // Your authentication library

export async function createUser(
  data: { name: string; email: string },
  token: string
) {
  const user = authenticate(token)
  if (!user) {
    throw new Error('Unauthorized')
  }
  const newUser = await db.user.create({ data })
```

```
  return newUser
}

'use server'

import { db } from '@/lib/db' // Your database client
import { authenticate } from '@/lib/auth' // Your authentication library

export async function createUser(data, token) {
  const user = authenticate(token)
  if (!user) {
    throw new Error('Unauthorized')
  }
  const newUser = await db.user.create({ data })
  return newUser
}
```

Reference

See the React documentation for more information on `use server`.

---

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	-
`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: Learn how to use the `<Form>` component to handle form submissions and search params updates with client-side navigation.

The `<Form>` component extends the HTML `<form>` element to provide **prefetching** of loading UI, **client-side navigation** on submission, and **progressive enhancement**.

It's useful for forms that update URL search params as it reduces the boilerplate code needed to achieve the above.

Basic usage:

```
import Form from 'next/form'

export default function Page() {
  return (
    <Form action="/search">
      {/* On submission, the input value will be appended to
          the URL, e.g. /search?query=abc */}
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

```
import Form from 'next/form'

export default function Search() {
  return (
    <Form action="/search">
      {/* On submission, the input value will be appended to
          the URL, e.g. /search?query=abc */}
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

```
import Form from 'next/form'

export default function Page() {
  return (
    <Form action="/search">
```

```
      {/* On submission, the input value will be appended to
          the URL, e.g. /search?query=abc */}
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

```
import Form from 'next/form'

export default function Search() {
  return (
    <Form action="/search">
      {/* On submission, the input value will be appended to
          the URL, e.g. /search?query=abc */}
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

Reference

The behavior of the `<Form>` component depends on whether the `action` prop is passed a `string` or `function`.

- When `action` is a **string**, the `<Form>` behaves like a native HTML form that uses a **`GET`** method. The form data is encoded into the URL as search params, and when the form is submitted, it navigates to the specified URL. In addition, Next.js:
  - Prefetches the path when the form becomes visible, this preloads shared UI (e.g. `layout.js` and `loading.js`), resulting in faster navigation.
  - Performs a client-side navigation instead of a full page reload when the form is submitted. This retains shared UI and client-side state.
- When `action` is a **function** (Server Action), `<Form>` behaves like a React form, executing the action when the form is submitted.

- When `action` is a **string**, the `<Form>` behaves like a native HTML form that uses a **`GET`** method. The form data is encoded into the URL as search params, and when the form is submitted, it navigates to the specified URL. In addition, Next.js:
  - Performs a client-side navigation instead of a full page reload when the form is submitted. This retains shared UI and client-side state.

`action` (string) Props

When `action` is a string, the `<Form>` component supports the following props:

Prop	Example	Type	Required
`action`	`action="/search"`	`string` (URL or relative path)	Yes
`replace`	`replace={false}`	`boolean`	-
`scroll`	`scroll={true}`	`boolean`	-

- **`action`**: The URL or path to navigate to when the form is submitted.
  - An empty string `""` will navigate to the same route with updated search params.
- **`replace`**: Replaces the current history state instead of pushing a new one to the browser's history stack. Default is `false`.
- **`scroll`**: Controls the scroll behavior during navigation. Defaults to `true`, this means it will scroll to the top of the new route, and maintain the scroll position for backwards and forwards navigation.

When `action` is a string, the `<Form>` component supports the following props:

Prop	Example	Type	Required
`action`	`action="/search"`	`string` (URL or relative path)	Yes
`replace`	`replace={false}`	`boolean`	-
`scroll`	`scroll={true}`	`boolean`	-
`prefetch`	`prefetch={true}`	`boolean`	-

- **`action`**: The URL or path to navigate to when the form is submitted.
  - An empty string `""` will navigate to the same route with updated search params.
- **`replace`**: Replaces the current history state instead of pushing a new one to the browser's history stack. Default is `false`.
- **`scroll`**: Controls the scroll behavior during navigation. Defaults to `true`, this means it will scroll to the top of the new route, and maintain the scroll position for backwards and forwards navigation.

- **`prefetch`**: Controls whether the path should be prefetched when the form becomes visible in the user's viewport. Defaults to `true`.

`action` (function) Props

When `action` is a function, the `<Form>` component supports the following prop:

Prop	Example	Type	Required
`action`	`action={myAction}`	`function` (Server Action)	Yes

- **`action`**: The Server Action to be called when the form is submitted. See the React docs for more.

  Good to know: When `action` is a function, the `replace` and `scroll` props are ignored.

Caveats

- **`formAction`**: Can be used in a `<button>` or `<input type="submit">` fields to override the `action` prop. Next.js will perform a client-side navigation, however, this approach doesn't support prefetching.
  - When using `basePath`, you must also include it in the `formAction` path. e.g. `formAction="/base-path/search"`.
- **`key`**: Passing a `key` prop to a string `action` is not supported. If you'd like to trigger a re-render or perform a mutation, consider using a function `action` instead.

- **`onSubmit`**: Can be used to handle form submission logic. However, calling `event.preventDefault()` will override `<Form>` behavior such as navigating to the specified URL.
- **`method`**, **`encType`**, **`target`**: Are not supported as they override `<Form>` behavior.
  - Similarly, `formMethod`, `formEncType`, and `formTarget` can be used to override the `method`, `encType`, and `target` props respectively, and using them will fallback to native browser behavior.
  - If you need to use these props, use the HTML `<form>` element instead.
- **`<input type="file">`**: Using this input type when the `action` is a string will match browser behavior by submitting the filename instead of the file object.

Examples

Search form that leads to a search result page

You can create a search form that navigates to a search results page by passing the path as an `action`:

```
import Form from 'next/form'

export default function Page() {
  return (
    <Form action="/search">
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

```
import Form from 'next/form'

export default function Page() {
  return (
    <Form action="/search">
      <input name="query" />
      <button type="submit">Submit</button>
    </Form>
  )
}
```

When the user updates the query input field and submits the form, the form data will be encoded into the URL as search params, e.g. `/search?query=abc`.

> **Good to know**: If you pass an empty string `""` to `action`, the form will navigate to the same route with updated search params.

On the results page, you can access the query using the `searchParams` `page.js` prop and use it to fetch data from an external source.

```
import { getSearchResults } from '@/lib/search'

export default async function SearchPage({
  searchParams,
}: {
  searchParams: { [key: string]: string | string[] | undefined }
}) {
  const results = await getSearchResults(searchParams.query)

  return <div>...</div>
}
```

```
import { getSearchResults } from '@/lib/search'

export default async function SearchPage({ searchParams }) {
  const results = await getSearchResults(searchParams.query)

  return <div>...</div>
}
```

When the `<Form>` becomes visible in the user's viewport, shared UI (such as `layout.js` and `loading.js`) on the `/search` page will be prefetched. On submission, the form will immediately navigate to the new route and show loading UI while the results are being fetched. You can design the fallback UI using `loading.js`:

```
export default function Loading() {
  return <div>Loading...</div>
}
```

```
export default function Loading() {
  return <div>Loading...</div>
}
```

To cover cases when shared UI hasn't yet loaded, you can show instant feedback to the user using `useFormStatus`.

First, create a component that displays a loading state when the form is pending:

```
'use client'
import { useFormStatus } from 'react-dom'

export default function SearchButton() {
  const status = useFormStatus()
  return (
    <button type="submit">{status.pending ? 'Searching...' : 'Search'}</button>
  )
}
```

```
'use client'
import { useFormStatus } from 'react-dom'

export default function SearchButton() {
  const status = useFormStatus()
  return (
    <button type="submit">{status.pending ? 'Searching...' : 'Search'}</button>
  )
}
```

Then, update the search form page to use the `SearchButton` component:

```
import Form from 'next/form'
import { SearchButton } from '@/ui/search-button'

export default function Page() {
  return (
    <Form action="/search">
      <input name="query" />
      <SearchButton />
    </Form>
  )
}
```

```
import Form from 'next/form'
import { SearchButton } from '@/ui/search-button'

export default function Page() {
  return (
    <Form action="/search">
      <input name="query" />
      <SearchButton />
    </Form>
  )
}
```

Mutations with Server Actions

You can perform mutations by passing a function to the `action` prop.

```
import Form from 'next/form'
import { createPost } from '@/posts/actions'

export default function Page() {
  return (
    <Form action={createPost}>
      <input name="title" />
      {/* ... */}
      <button type="submit">Create Post</button>
    </Form>
  )
}
```

```
import Form from 'next/form'
import { createPost } from '@/posts/actions'

export default function Page() {
  return (
    <Form action={createPost}>
      <input name="title" />
      {/* ... */}
      <button type="submit">Create Post</button>
    </Form>
  )
}
```

After a mutation, it's common to redirect to the new resource. You can use the `redirect` function from `next/navigation` to navigate to the new post page.

> **Good to know**: Since the "destination" of the form submission is not known until the action is executed, `<Form>` cannot automatically prefetch shared UI.

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

export async function createPost(formData: FormData) {
  // Create a new post
  // ...

  // Redirect to the new post
  redirect(`/posts/${data.id}`)
}
```

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

export async function createPost(formData) {
  // Create a new post
  // ...

  // Redirect to the new post
  redirect(`/posts/${data.id}`)
}
```

Then, in the new page, you can fetch data using the `params` prop:

```
import { getPost } from '@/posts/data'

export default async function PostPage({
  params,
}: {
  params: Promise<{ id: string }>
}) {
  const data = await getPost((await params).id)

  return (
    <div>
      <h1>{data.title}</h1>
      {/* ... */}
    </div>
  )
}
```

```
import { getPost } from '@/posts/data'

export default async function PostPage({ params }) {
  const data = await getPost((await params).id)

  return (
    <div>
      <h1>{data.title}</h1>
      {/* ... */}
    </div>
  )
}
```

See the Server Actions docs for more examples.

---

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 | - | | [`overrideSrc`](#overridesrc) | `overrideSrc="/seo.png"` | String | - |

Required Props

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

```
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 the default loader, also consider the following for source images:

- When src is an external URL, you must also configure remotePatterns
- When src is animated or not a known format (JPEG, PNG, WebP, AVIF, GIF, TIFF) the image will be served as-is
- When src is SVG format, it will be blocked unless `unoptimized` or `dangerouslyAllowSVG` is enabled

`width`

The `width` property represents the *intrinsic* image width in pixels. This property is used to infer the correct aspect ratio of the image and avoid layout shift during loading. It does not determine the rendered size of the image, which is controlled by CSS, similar to the `width` attribute in the HTML `<img>` tag.

Required, except for statically imported images or images with the `fill` property.

`height`

The `height` property represents the *intrinsic* image height in pixels. This property is used to infer the correct aspect ratio of the image and avoid layout shift during loading. It does not determine the rendered size of the image, which is controlled by CSS, similar to the `height` attribute in the HTML `<img>` tag.

Required, except for statically imported images or images with the `fill` property.

> **Good to know:**
>
> - Combined, both `width` and `height` properties are used to determine the aspect ratio of the image which used by browsers to reserve space for the image before it loads.
> - The intrinsic size does not always mean the rendered size in the browser, which will be determined by the parent container. For example, if the parent container is smaller than the intrinsic size, the image will be scaled down to fit the container.
> - You can use the `fill` property when the width and height are unknown.

`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.

```
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}
    />
  )
}
```

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 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`. If the `loading` property is also used and set to `lazy`, the `priority` property can't be used. The `loading` property is only meant for advanced use cases. Remove `loading` when `priority` is needed.

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)} />

<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`

```
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,
  },
}
```

`overrideSrc`

When providing the `src` prop to the `<Image>` component, both the `srcset` and `src` attributes are generated automatically for the resulting `<img>`.

```
<Image src="/me.jpg" />

<img
  srcset="
    /_next/image?url=%2Fme.jpg&w=640&q=75 1x,
    /_next/image?url=%2Fme.jpg&w=828&q=75 2x
  "
  src="/_next/image?url=%2Fme.jpg&w=828&q=75"
/>
```

In some cases, it is not desirable to have the `src` attribute generated and you may wish to override it using the `overrideSrc` prop.

For example, when upgrading an existing website from `<img>` to `<Image>`, you may wish to maintain the same `src` attribute for SEO purposes such as image ranking or avoiding recrawl.

```
<Image src="/me.jpg" overrideSrc="/override.jpg" />

<img
  srcset="
    /_next/image?url=%2Fme.jpg&w=640&q=75 1x,
    /_next/image?url=%2Fme.jpg&w=828&q=75 2x
  "
  src="/override.jpg"
/>
```

decoding

A hint to the browser indicating if it should wait for the image to be decoded before presenting other content updates or not. Defaults to `async`.

Possible values are the following:

- `async` - Asynchronously decode the image and allow other content to be rendered before it completes.
- `sync` - Synchronously decode the image for atomic presentation with other content.
- `auto` - No preference for the decoding mode; the browser decides what's best.

Learn more about the `decoding` attribute.

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.

Configuration Options

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

`localPatterns`

You can optionally configure `localPatterns` in your `next.config.js` file in order to allow specific paths to be optimized and block all others paths.

```
module.exports = {
  images: {
    localPatterns: [
      {
        pathname: '/assets/images/**',
        search: '',
      },
    ],
  },
}
```

> **Good to know**: The example above will ensure the `src` property of `next/image` must start with `/assets/images/` and must not have a query string. Attempting to optimize any other path will respond with 400 Bad Request.

`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/**',
        search: '',
      },
    ],
  },
}
```

> **Good to know**: The example above will ensure the `src` property of `next/image` must start with `https://example.com/account123/` and must not have a query string. Any other protocol, hostname, port, or unmatched path will respond with 400 Bad Request.

Below is an example of the `remotePatterns` property in the `next.config.js` file using a wildcard pattern in the `hostname`:

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
        port: '',
        search: '',
      },
    ],
  },
}
```

> **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. It cannot have a port or query string. Any other protocol or unmatched hostname will respond with 400 Bad Request.

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

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

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

> **Good to know**: When omitting `protocol`, `port`, `pathname`, or `search` then the wildcard `**` is implied. This is not recommended because it may allow malicious actors to optimize urls you did not intend.

Below is an example of the `remotePatterns` property in the `next.config.js` file using `search`:

```
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'assets.example.com',
        search: '?v=1727111025337',
      },
    ],
  },
}
```

> **Good to know**: The example above will ensure the `src` property of `next/image` must start with `https://assets.example.com` and must have the exact query string `?v=1727111025337`. Any other protocol or query string will respond with 400 Bad Request.

`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}`
}

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 in order to determine the best output format.

If the `Accept` header 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 and still fallback to WebP with the following configuration.

```
module.exports = {
  images: {
    formats: ['image/avif', 'image/webp'],
  },
}
```

Good to know:

- AVIF generally takes 50% 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 proper Content Security Policy (CSP) headers.

Therefore, we recommended using the `unoptimized` prop when the `src` prop is known to be SVG. This happens automatically when `src` ends with `".svg"`.

However, 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.

`contentDispositionType`

The default loader sets the `Content-Disposition` header to `attachment` for added protection since the API can serve arbitrary remote images.

The default value is `attachment` which forces the browser to download the image when visiting directly. This is particularly important when `dangerouslyAllowSVG` is true.

You can optionally configure `inline` to allow the browser to render the image when visiting directly, without downloading it.

```
module.exports = {
  images: {
    contentDispositionType: 'inline',
  },
}
```

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: '300px', height: '500px' }}>
      <Image
        src={photoUrl}
        alt="Picture of the author"
        sizes="300px"
        fill
        style={{
          objectFit: 'contain',
        }}
      />
    </div>
  )
}
```

Try it out:

- Demo the `fill` prop

Theme Detection CSS

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

getImageProps

For more advanced use cases, you can call `getImageProps()` to get the props that would be passed to the underlying `<img>` element, and instead pass to them to another component, style, canvas, etc.

This also avoid calling React `useState()` so it can lead to better performance, but it cannot be used with the `placeholder` prop because the placeholder will never be removed.

Theme Detection Picture

If you want to display a different image for light and dark mode, you can use the `<picture>` element to display a different image based on the user's preferred color scheme.

```
import { getImageProps } from 'next/image'

export default function Page() {
  const common = { alt: 'Theme Example', width: 800, height: 400 }
  const {
    props: { srcSet: dark },
  } = getImageProps({ ...common, src: '/dark.png' })
  const {
    props: { srcSet: light, ...rest },
  } = getImageProps({ ...common, src: '/light.png' })

  return (
    <picture>
      <source media="(prefers-color-scheme: dark)" srcSet={dark} />
      <source media="(prefers-color-scheme: light)" srcSet={light} />
      <img {...rest} />
    </picture>
  )
}
```

Art Direction

If you want to display a different image for mobile and desktop, sometimes called Art Direction, you can provide different `src`, `width`, `height`, and `quality` props to `getImageProps()`.

```
import { getImageProps } from 'next/image'

export default function Home() {
  const common = { alt: 'Art Direction Example', sizes: '100vw' }
  const {
    props: { srcSet: desktop },
  } = getImageProps({
    ...common,
    width: 1440,
    height: 875,
    quality: 80,
    src: '/desktop.jpg',
  })
  const {
    props: { srcSet: mobile, ...rest },
  } = getImageProps({
    ...common,
    width: 750,
    height: 1334,
    quality: 70,
    src: '/mobile.jpg',
  })

  return (
    <picture>
      <source media="(min-width: 1000px)" srcSet={desktop} />
      <source media="(min-width: 500px)" srcSet={mobile} />
      <img {...rest} style={{ width: '100%', height: 'auto' }} />
    </picture>
  )
}
```

Background CSS

You can even convert the `srcSet` string to the `image-set()` CSS function to optimize a background image.

```
import { getImageProps } from 'next/image'

function getBackgroundImage(srcSet = '') {
  const imageSet = srcSet
    .split(', ')
    .map((str) => {
```

```
      const [url, dpi] = str.split(' ')
      return `url("${url}") ${dpi}`
    })
    .join(', ')
  return `image-set(${imageSet})`
}

export default function Home() {
  const {
    props: { srcSet },
  } = getImageProps({ alt: '', width: 128, height: 128, src: '/img.png' })
  const backgroundImage = getBackgroundImage(srcSet)
  const style = { height: '100vh', width: '100vw', backgroundImage }

  return (
    <main style={style}>
      <h1>Hello World</h1>
    </main>
  )
}
```

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
`v15.0.0`	`decoding` prop added. `contentDispositionType` configuration default changed to `attachment`.
`v14.2.0`	`overrideSrc` prop added.
`v14.1.0`	`getImageProps()` is stable.
`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. */}

`<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.

Basic usage:

```
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>
}
```

```
import Link from 'next/link'

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


```
import Link from 'next/link'

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


Reference

The following props can be passed to 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	-
`legacyBehavior`	`legacyBehavior={true}`	Boolean	-
`passHref`	`passHref={true}`	Boolean	-
`shallow`	`shallow={false}`	Boolean	-
`locale`	`locale="fr"`	String or Boolean	-

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

> **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.

```
import Link from 'next/link'

// Navigate to /about?name=test
export default function Page() {
  return (
    <Link
      href={{
        pathname: '/about',
        query: { name: 'test' },
      }}
    >
      About
    </Link>
  )
}
```


```
import Link from 'next/link'

// Navigate to /about?name=test
export default function Page() {
  return (
    <Link
      href={{
        pathname: '/about',
        query: { name: 'test' },
      }}
    >
      About
    </Link>
  )
}
```


```
import Link from 'next/link'

// Navigate to /about?name=test
export default function Home() {
  return (
    <Link
      href={{
        pathname: '/about',
        query: { name: 'test' },
      }}
    >
      About
    </Link>
  )
}
```


```
import Link from 'next/link'

// Navigate to /about?name=test
export default function Home() {
  return (
    <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>
  )
}
```

```
import Link from 'next/link'

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

```
import Link from 'next/link'

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

`scroll`

Defaults to `true`. The default scrolling behavior of `<Link>` in Next.js **is to maintain scroll position**, similar to how browsers handle back and forwards navigation. When you navigate to a new Page, scroll position will stay the same as long as the Page is visible in the viewport. However, if the Page is not visible in the viewport, Next.js will scroll to the top of the first Page element.

When `scroll = {false}`, Next.js will not attempt to scroll to the first Page element.

> **Good to know**: Next.js checks if `scroll: false` before managing scroll behavior. If scrolling is enabled, it identifies the relevant DOM node for navigation and inspects each top-level element. All non-scrollable elements and those without rendered HTML are bypassed, this includes sticky or fixed positioned elements, and non-visible elements such as those calculated with `getBoundingClientRect`. Next.js then continues through siblings until it identifies a scrollable element that is visible in the viewport.

```
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>
  )
}
```

```
import Link from 'next/link'

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

```
import Link from 'next/link'

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

`prefetch`

Prefetching happens when a `<Link />` component enters the user's viewport (initially or through scroll). Next.js prefetches and loads the linked route (denoted by the `href`) and its data in the background to improve the performance of client-side navigations. If the prefetched data has expired by the time the user hovers over a `<Link />`, Next.js will attempt to prefetch it again. **Prefetching is only enabled in production**.

The following values can be passed to the `prefetch` prop:

- **`null` (default)**: Prefetch behavior depends on whether the route is static or dynamic. For static routes, the full route will be prefetched (including all its data). For dynamic routes, the partial route down to the nearest segment with a `loading.js` boundary will be prefetched.
- `true`: The full route will be prefetched for both static and dynamic routes.
- `false`: Prefetching will never happen both on entering the viewport and on hover.

```
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>
  )
}
```


Prefetching happens when a `<Link />` component enters the user's viewport (initially or through scroll). Next.js prefetches and loads the linked route (denoted by the `href`) and data in the background to improve the performance of client-side navigation's. **Prefetching is only enabled in production**.

The following values can be passed to the `prefetch` prop:

- **`true` (default)**: The full route and its data will be prefetched.
- `false`: Prefetching will not happen when entering the viewport, but will happen on hover. If you want to completely remove fetching on hover as well, consider using an `<a>` tag or incrementally adopting the App Router, which enables disabling prefetching on hover too.

```
import Link from 'next/link'

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


```
import Link from 'next/link'

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


`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`. See the passing a functional component example for more information.

`scroll`

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

```
import Link from 'next/link'

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


```
import Link from 'next/link'

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


`shallow`

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

```
import Link from 'next/link'

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


```
import Link from 'next/link'

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

`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.

```
import Link from 'next/link'

export default function Home() {
  return (
    <>
      {/* Default behavior: locale is prepended */}
      <Link href="/dashboard">Dashboard (with locale)</Link>

      {/* Disable locale prepending */}
      <Link href="/dashboard" locale={false}>
        Dashboard (without locale)
      </Link>

      {/* Specify a different locale */}
      <Link href="/dashboard" locale="fr">
        Dashboard (French)
      </Link>
    </>
  )
}
```

```
import Link from 'next/link'

export default function Home() {
  return (
    <>
      {/* Default behavior: locale is prepended */}
      <Link href="/dashboard">Dashboard (with locale)</Link>

      {/* Disable locale prepending */}
      <Link href="/dashboard" locale={false}>
        Dashboard (without locale)
      </Link>

      {/* Specify a different locale */}
      <Link href="/dashboard" locale="fr">
        Dashboard (French)
      </Link>
    </>
  )
}
```

Examples

The following examples demonstrate how to use the `<Link>` component in different scenarios.

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'

interface Post {
  id: number
  title: string
  slug: string
}

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

```
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>
      <Link className={`link ${pathname === '/' ? 'active' : ''}`} href="/">
        Home
      </Link>

      <Link
```

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

```
'use client'

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

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

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

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

Scrolling to an `id`

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>
```

> **Good to know**:
>
> - Next.js will scroll to the Page if it is not visible in the viewport upon navigation.

Linking to dynamic route segments

For dynamic route segments, 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>
  )
}
```

```
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'

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

```
import Link from 'next/link'

export default 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
```

```
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.

Nesting 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'
import React from 'react'

// Define the props type for MyButton
interface MyButtonProps {
  onClick?: React.MouseEventHandler<HTMLAnchorElement>
  href?: string
}

// Use React.ForwardRefRenderFunction to properly type the forwarded ref
const MyButton: React.ForwardRefRenderFunction<
  HTMLAnchorElement,
  MyButtonProps
> = ({ onClick, href }, ref) => {
  return (
    <a href={href} onClick={onClick} ref={ref}>
      Click Me
    </a>
  )
}

// Use React.forwardRef to wrap the component
const ForwardedMyButton = React.forwardRef(MyButton)

export default function Page() {
  return (
    <Link href="/about" passHref legacyBehavior>
      <ForwardedMyButton />
    </Link>
  )
}
```

```
import Link from 'next/link'
import React from 'react'

// `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>
  )
})

// Add a display name for the component (useful for debugging)
MyButton.displayName = 'MyButton'

export default function Page() {
  return (
    <Link href="/about" passHref legacyBehavior>
```

```
      <MyButton />
    </Link>
  )
}
```

```
import Link from 'next/link'
import React from 'react'

// Define the props type for MyButton
interface MyButtonProps {
  onClick?: React.MouseEventHandler<HTMLAnchorElement>
  href?: string
}

// Use React.ForwardRefRenderFunction to properly type the forwarded ref
const MyButton: React.ForwardRefRenderFunction<
  HTMLAnchorElement,
  MyButtonProps
> = ({ onClick, href }, ref) => {
  return (
    <a href={href} onClick={onClick} ref={ref}>
      Click Me
    </a>
  )
}

// Use React.forwardRef to wrap the component
const ForwardedMyButton = React.forwardRef(MyButton)

export default function Home() {
  return (
    <Link href="/about" passHref legacyBehavior>
      <ForwardedMyButton />
    </Link>
  )
}
```

```
import Link from 'next/link'
import React from 'react'

// `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>
  )
})

// Add a display name for the component (useful for debugging)
MyButton.displayName = 'MyButton'

export default function Home() {
  return (
    <Link href="/about" passHref legacyBehavior>
      <MyButton />
    </Link>
  )
}
```

Passing a URL Object

`Link` can also receive a URL object and it will automatically format it to create the URL string:

```
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
```

```
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:

```
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/about" replace>
      About us
    </Link>
  )
}
```


```
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/about" replace>
      About us
    </Link>
  )
}
```


```
import Link from 'next/link'

export default function Home() {
  return (
    <Link href="/about" replace>
      About us
    </Link>
  )
}
```


```
import Link from 'next/link'

export default function Home() {
  return (
    <Link href="/about" replace>
      About us
    </Link>
  )
}
```


Disable scrolling to the top of the page

The default scrolling behavior of `<Link>` in Next.js **is to maintain scroll position**, similar to how browsers handle back and forwards navigation. When you navigate to a new Page, scroll position will stay the same as long as the Page is visible in the viewport.

However, if the Page is not visible in the viewport, Next.js will scroll to the top of the first Page element. 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()`.

```
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/#hashid" scroll={false}>
      Disables scrolling to the top
    </Link>
  )
}
```


```
import Link from 'next/link'

export default function Page() {
  return (
    <Link href="/#hashid" scroll={false}>
      Disables scrolling to the top
    </Link>
  )
}
```


Using `router.push()` or `router.replace()`:

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

const router = useRouter()

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

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`:

```
import Link from 'next/link'

export default function Home() {
  return (
    <Link href="/#hashid" scroll={false}>
      Disables scrolling to the top
    </Link>
  )
}
```

```
import Link from 'next/link'

export default function Home() {
  return (
    <Link href="/#hashid" scroll={false}>
      Disables scrolling to the top
    </Link>
  )
}
```

Prefetching links in 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 can add the following in your Middleware to redirect the user to the correct page:

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

export function middleware(request: Request) {
  const nextUrl = request.nextUrl
  if (nextUrl.pathname === '/dashboard') {
    if (request.cookies.authToken) {
      return NextResponse.rewrite(new URL('/auth/dashboard', request.url))
    } else {
      return NextResponse.rewrite(new URL('/public/dashboard', request.url))
    }
  }
}
```

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

export function middleware(request) {
  const nextUrl = request.nextUrl
  if (nextUrl.pathname === '/dashboard') {
    if (request.cookies.authToken) {
      return NextResponse.rewrite(new URL('/auth/dashboard', request.url))
    } else {
      return NextResponse.rewrite(new URL('/public/dashboard', request.url))
    }
  }
}
```

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

```
'use client'

import Link from 'next/link'
import useIsAuthed from './hooks/useIsAuthed' // Your auth hook

export default function Page() {
  const isAuthed = useIsAuthed()
  const path = isAuthed ? '/auth/dashboard' : '/public/dashboard'
  return (
    <Link as="/dashboard" href href={path}>
      Dashboard
    </Link>
  )
}
```

```
'use client'

import Link from 'next/link'
import useIsAuthed from './hooks/useIsAuthed' // Your auth hook

export default function Page() {
  const isAuthed = useIsAuthed()
  const path = isAuthed ? '/auth/dashboard' : '/public/dashboard'
  return (
    <Link as="/dashboard" href={path}>
      Dashboard
    </Link>
  )
}
```

```
'use client'

import Link from 'next/link'
import useIsAuthed from './hooks/useIsAuthed' // Your auth hook

export default function Home() {
  const isAuthed = useIsAuthed()
  const path = isAuthed ? '/auth/dashboard' : '/public/dashboard'
  return (
    <Link as="/dashboard" href href={path}>
      Dashboard
    </Link>
  )
}
```

```
'use client'

import Link from 'next/link'
import useIsAuthed from './hooks/useIsAuthed' // Your auth hook

export default function Home() {
  const isAuthed = useIsAuthed()
  const path = isAuthed ? '/auth/dashboard' : '/public/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/authed/[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: